Moving code analysis from safety to security : attacker model

Soline Ducousso

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : VERIMAG

Aller de la sûreté à la sécurité en analyse de code : le modèle d'attaquant

Moving code analysis from safety to security: attacker model

Présentée par :

Soline DUCOUSSO

Direction de thèse :

Marie laure POTET PROFESSEURE DES UNIVERSITES, GRENOBLE INP	Directrice de thèse
Sébastien BARDIN INGENIEUR DE RECHERCHE, CEA DE PARIS-SACLAY	Co-encadrant de thèse

Rapporteurs :

KARINE HEYDEMANN
INGENIEURE HDR, THALES RESEARCH & TECHNOLOGY
GUILLAUME HIET
PROFESSEUR, CENTRALESUPELEC RENNES

Thèse soutenue publiquement le **14 décembre 2023**, devant le jury composé de :

MARIE-LAURE POTET PROFESSEURE DES UNIVERSITES, GRENOBLE INP	Directrice de thèse
KARINE HEYDEMANN INGENIEURE HDR, THALES RESEARCH & TECHNOLOGY	Rapporteure
GUILLAUME HIET PROFESSEUR, CENTRALESUPELEC RENNES	Rapporteur
LAURE GONNORD PROFESSEURE DES UNIVERSITES, GRENOBLE INP	Présidente
GUILLAUME BOUFFARD INGENIEUR DE RECHERCHE, ANSSI	Examinateur

Invités :

SEBASTIEN BARDIN
INGENIEUR DE RECHERCHE, CEA DE PARIS-SACLAY

Résumé

Aller de la Sûreté à la Sécurité en Analyse de Code: le Modèle d'Attaquant

Des travaux majeurs ont été réalisés, ces dernières décennies, dans le domaine de l'analyse de programme, tirant parti de techniques telles que l'exécution symbolique, l'analyse statique, l'interprétation abstraite ou la vérification bornée de modèles, pour chasser les bogues et vulnérabilités logicielles, conduisant à l'adoption de ces techniques par de grandes entreprises. Les bogues étant des points d'entrées pour les attaques, les éliminer est un premier pas vers une meilleure sécurité logicielle. Cependant, ces techniques reposent sur un modèle d'attaquant seulement capable de concevoir des entrées malicieuses, exploitant des cas particuliers dans le code lui-même, typiquement un fichier formaté de manière à déclencher un débordement mémoire dans les routines de traitement. Un modèle d'attaquant plutôt faible, alors qu'un attaquant avancé est capable de perturber l'exécution d'un programme en exploitant des vecteurs d'attaques tels que les injections de fautes matérielles, les attaques micro-architecturales, les attaques matérielles contrôlées par logiciel, et n'importe quelle combinaison de ces vecteurs d'attaques.

Dans cette thèse, notre objectif est de concevoir une technique automatique et efficace pour raisonner sur l'impact d'un attaquant avancé sur les propriétés de sécurité d'un programme. Nous proposons l'atteignabilité adversariale, un formalisme qui étend la notion d'atteignabilité en y incluant les capacités de l'attaquant. Nous avons construit un nouvel algorithme, l'exécution symbolique adversariale, pour répondre au problème de l'atteignabilité adversariale sous l'angle de la recherche de bogues (vérification bornée). Notre algorithme évite l'augmentation significative de l'espace d'états à analyser due nouvelles capacités de l'attaquant, grâce à un nouvel encodage non branchant de celles-ci sous forme de fautes injectées. Nous le montrons correct et k-complet pour l'atteignabilité adversariale. De plus, nous avons imaginé deux optimisations visant à réduire le nombre de fautes : la détection précoce de saturation et l'injection à la demande.

Nous proposons une implémentation de l'exécution symbolique adversariale au niveau binaire intégrée à l'outil BINSEC. Notre évaluation expérimentale repose sur des programmes usuels du domaine des fautes matérielles et des cartes à puce. Nos expériences montrent un gain de performance significatif par rapport à l'état de l'art, en moyenne d'un facteur x20 et x400 pour 1 et 2 fautes respectivement, avec un gain similaire en termes de chemins explorés. De plus, notre approche passe à l'échelle jusqu'à considérer 10 fautes alors que l'état de l'art atteint le temps limite pour 3 fautes. Nous montrons également l'utilisabilité de notre méthode dans différents scénarios de sécurité tels que reproduire une attaque BellCoRe sur CRT-RSA, chercher des attaques sur un programme protégé par la contre-mesure SecSwift, ou évaluer la robustesse d'un réseau de neurones. D'autre part, nous avons exploré le cas du programme de démarrage de WooKey en rejouant des attaques et en évaluant des contre-mesures proposées. En particulier, nous avons trouvé une attaque non rapportée précédemment et avons proposé un nouveau correctif aux développeurs.

Mots-clefs: sécurité logicielle · méthodes formelles · analyse de code · code binaire · vulnérabilités.

Abstract

Moving Code Analysis from Safety to Security: the Attacker Model

Major works have delved into program analysis over the last decades, leveraging techniques such as symbolic execution, static analysis, abstract interpretation or bounded model checking, to hunt for software vulnerabilities and bugs in programs, or to prove their absence, leading to industrial adoption in some leading companies. As bugs are an attack entry point, removing them is a first step towards better software security. Yet, they are based on the standard concept of reachability and represent an attacker able to craft smart, legitimate input, through legitimate input sources of the program, exploiting corner cases in the code itself, a rather weak threat model, typically a file formatted to trigger a buffer overflow in data processing. Tools only looking for bugs and software vulnerabilities may deem a program secure while the bar remains quite low for an advanced attacker, able for example to take advantage of attack vectors such as (physical) hardware fault injections, micro-architectural attacks, software-based hardware attacks like Rowhammer, or any combination of vectors.

In this thesis, our goal is to devise a technique to automatically and efficiently reason about the impact of an advanced attacker onto a program security properties. We propose adversarial reachability, a formalism extending standard reachability to reason about a program execution in the presence of an advanced attacker. As equipping the attacker with new capabilities significantly increases the state space of the program under analysis, we build a new algorithm based on symbolic techniques, named adversarial symbolic execution, to address the adversarial reachability problem from the bug-finding point of view (bounded verification). Our algorithm prevents path explosion thanks to a new forkless encoding of attacker capabilities, modeled as faults. We show it correct and k-complete with respect to adversarial reachability. To improve the performance further, we design two new optimizations to reduce the number of injected faults while keeping the same attacker power: Early Detection of fault Saturation and Injection On Demand.

We propose an implementation of our techniques for binary-level analysis, on top of the BINSEC framework. We systematically evaluate its performances against prior work, using a standard fault injection benchmark from physical fault attacks and smart cards. Experiments show a very significant performance gain against prior approaches, for example up to x20 and x400 times on average for 1 and 2 faults respectively - with a similar reduction in the number of adversarial paths. Moreover, our approach scales up to 10 faults whereas the state-of-the-art starts to timeout for 3 faults. We also show the use of our method in a number of different security scenarios such as reproducing a BellCoRe attack with one reset fault on CRT-RSA or looking for attacks on a program protected by the SecSwift countermeasure. In addition, we perform a security analysis of the well-tested WooKey bootloader and demonstrate the ability of our analysis to find attacks and evaluate countermeasures in real-life security scenarios. We were especially able to find an attack not reported before on a recently proposed patch, and proposed a new patch to the developers.

Keywords: software security · formal methods · code analysis · binary code · vulnerabilities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Context

Programs. *Programs* are everywhere in our modern lives: our phones in our pockets, the computers we work on, our TVs at home, our local grocery store systems, but also the chips in our credit cards, systems in our cars, buses, trains, boats and planes, our hospital system, our water delivery system and our power grids, etc. Programs receive input from the world (or us), then perform a function and have an output, either as new data stored somewhere, as information display or as commands for actuators. Some features can be *security-oriented*, meaning that their failure can have important consequences, material-, money-, human- or information-wise.

Vulnerabilities. Because programs are human-written, some errors are made. The specification for the developers can have holes or inconsistencies. The developers can make mistakes in their development work, or make incorrect assumptions about libraries, interpreters or compilers behaviors. This leaves programs with *bugs* that can be exploited. Program protections can be added to reduce the attack surface, but are often incomplete when faced with an advanced attacker. Furthermore, *advanced attackers* can not only leverage existing bugs present in a program but also perform active fault injection attacks, injecting faulty behavior directly into the execution of the program. New attack techniques are regularly discovered, increasing an advanced attacker's capabilities.

Attackers. Some entities have an interest in exploiting a program's incorrect behaviors. Malicious groups such as government agencies, mafias or companies are known for, or suspected of, cyber-attacks like ransomware, blackmail, theft and corporate espionage. But they are not the only ones testing programs. Script kiddies do it for fun and learning. Security researchers challenge programs and further the state-of-the-art knowledge in security. Companies test the security of their products with the help of auditors before market release.

Program Analysis. To prevent attackers from exploiting a program, major works have delved into *automated program analysis* over the last decades, leveraging techniques such as symbolic execution [CS13, GLM12, BGM13], static analysis [Fac], abstract interpretation [CGJ+03] or bounded model checking [CBRZ01], to hunt software vulnerabilities and bugs in programs, or to prove their absence [CCF+05, KKP+15], leading to industrial adoption in some leading companies [BGM13, Fac, BCLR04, KKP+15, LRV+22]. Ensuring a program works as intended, i.e. without bugs, is the notion of *safety*. As bugs are also an attack entry point, removing them ensures program safety and is a first step towards better software *security*.

1.2 Problem

Source Code VS Binary Analysis. Many program analysis techniques are performed at the source code level, to allow better developer adoption and understanding of the analysis results. However, what is actually executed by a computer is the binary program (for compiled languages), which can differ from the source code, binary code is where the real vulnerabilities lay. Binary analysis is a harder problem due to the loss of information such as types or data structures, and requires about ten times more lines than source code to express equivalent behavior.

Advanced Attackers Across Security Fields. In addition to bugs, a program has other weaknesses during its execution. The more resourceful attackers, which we call advanced attackers, are capable of using any attack vector to exploit a program, and perform multiple actions through the course of an attack. For instance, micro-architectural attacks misuse complex micro-architectural behaviors such as cache systems or branch prediction. The running hardware can even be physically disturbed with hardware fault injection means. Those complex vulnerabilities, exploitable by an advanced attacker, are rarely taken into account in automated program analysis techniques.

Security Progam Analysis. The existing program analysis techniques focus on finding bugs. It appears that all these methods consider a rather weak threat model, where the attacker can only craft smart "inputs of death" through legitimate input sources of the program, exploiting corner cases in the code itself. In order to perform a security evaluation, meaning replay advanced attacks at will, varying different parameters, perform security evaluation and program protection, there is a need for security program analysis that would combine automated program analysis and attacker model. Tools only looking for bugs and software vulnerabilities may deem a program secure while the bar remains quite low for an advanced attacker.

Efficient Progam Analysis Techniques. Taking into account all possible actions of an advanced attacker and their multiplicity in an analysis represents an added complexity, which leads to a state explosion in existing techniques. This limits the ability to replay attacks at will and to consider different attacker models.

1.3 Goal & Challenges

Our goal is to devise a technique to automatically and efficiently reason about the impact of an advanced attacker onto a program security properties, where the standard program analysis techniques only support an attacker crafting smart legitimate inputs.

Represent an Advanced Attacker in a Formal Framework. To represent the impact of an advanced attacker on a program, the first step is to devise a model of such an attacker. Then, we need to provide a *formal framework* to study what an advanced attacker can do to attack a program under study. Interestingly, while it has been little studied for program-level analysis, such frameworks are routinely used in cryptographic protocol verification [Cer01, BCL14]. This formal framework can then be used as a theoretical base for an automated program analysis technique for security.

Existing techniques tend to be domain specific and hence, represent an attacker usually able to exploit one attack vector, with only limited attack power. Our goal is to build a *generic model* able to represent an advanced attacker able to leverage multiple attack vectors.

Efficient and Generic Code Analysis Techniques for Security. The second challenge is to design an efficient algorithm to assess the vulnerability of a program to a given attacker model, when adding capabilities to the attacker naturally gives rise to a significant explosion of program states to consider – especially in the case of an attacker able to perform multiple actions.

The rare prior work in the field, mostly focused on encompassing physical fault injections for high-security devices, rely mostly on *mutant generation* [CCG13, RG14, GWJLL17, CDFG18, GWJL20] or *forking analysis* [PMPD14, BBC+14, BHE+19, LFBP21, Lan22], yielding scalability issues. Moreover, most of them are limited to a few predefined *fault models*, i.e. patterns of the injected faulty behavior, and do not propose a formalization of the underlying problem.

1.4 Proposal

Adversarial Reachability. We propose a model of an advanced attacker in terms of its capability to change program behavior and the goal of the attack as the violation of a security property of the program. We extend the usual transition system representing the execution of a program with new transitions modeling the attacker actions. Then, we extend the standard concept of reachability to reason about a program execution in the presence of an advanced attacker. *Adversarial reachability* is a formalism expressing the reachability of the attacker goal in this modified transition system.

Adversarial Symbolic Execution. We build a new algorithm based on symbolic techniques, named *adversarial symbolic execution* (ASE), to address the adversarial reachability problem from the bug-finding point of view (bounded verification). Our algorithm is generic in terms of supported attacker capabilities and prevents path explosion thanks to a new *forkless* encoding of attacker capabilities, represented as faults in the program. We show it correct and k-complete with respect to adversarial reachability. To improve the performance further, we design two new optimizations to reduce the number of injected faults: Early Detection of fault Saturation and Injection On Demand.

Tool Prototype: BINSEC/ASE. We implemented ASE by modifying an existing symbolic execution engine for binary programs, BINSEC, building BINSEC/ASE.

1.5 Contributions

As a summary, we claim the following contributions:

- We propose a model for a software-level advanced attacker and we formalize its impact on a program's security properties in Chapter 4. We named it the *adversarial reachability problem*, extending standard reachability to take into account an advanced attacker in a transition system augmented with transitions representing attacker actions. We define the associated correctness and completeness definitions;

- We describe a new symbolic exploration method, *adversarial symbolic execution*, to answer adversarial reachability in Chapter 4. The goal of this analysis technique is to automatically and efficiently reason about the impact of an advanced attacker on a program, and be generic in the supported attacker model. It features a novel forkless fault encoding to prevent path explosion and two optimization strategies to reduce the number of fault injections without loss of generality with respect to the attacker model. We establish adversarial symbolic execution correctness and completeness for adversarial reachability;

- We propose an implementation of adversarial symbolic execution for binary-level analysis, on top of the BINSEC framework [DBT+16] in a *tool called BINSEC/ASE*, described in Chapter 5. In Chapter 6, we systematically evaluate its performances against prior work, using a standard SWIFI benchmark from physical fault attacks and smart cards. We highlight the interest and feasibility of our technique by exploring different security scenarios. Finally, we perform a case study on the WooKey bootloader, with a vulnerability assessment of two implementations and we propose our own patch for an attack not reported before.

Publication

Part of the work presented in this thesis has been published in an paper at ESOP 2023: Soline Ducousso, Sébastien Bardin, and Marie-Laure Potet, "Adversarial Reachability for Program-level Security Analysis", *European Symposium on Programming*, 2023 [DBP23].

Talks & presentations

Part of this work has been presented at:

- *Journées du Groupe de Travail Méthodes Formelles pour la Sécurité* (GT MFS) in 2022 and 2023,

- the seminar *La Cybersécurité sur un plateau* (CoaP) in February 2023,

- *Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information* (RESSI) in 2023,

- *Journée thématique sur les attaques par injection de fautes* (JAIF) in 2023.

Software Package and Artefacts

Our benchmark infrastructure, case studies and the executable of BINSEC/ASE have been made available through artifacts for reproducibility purposes on GitHub[1] and Zenodo[2]. The BINSEC/ASE source code will be open-sourced.

1.6 Impact and Perspectives

This work is a first step in designing efficient program analysis techniques able to take into account advanced attackers. The approach is generic enough to accommodate many common fault models, including the bit flips from RowHammer, test inversion, instruction skip or arbitrary data modification; still, generic instruction modifications are currently out of reach. Also, while we investigate the bug-finding side of the problem (under-approximation), the verification side (over-approximation) is interesting as well. These are exciting directions for future research.

1.7 Manuscript Outline

The remainder of the manuscript is organized as follows:

- In Chapter 2, we describe the context of systems with security features and motivate the need for dedicated program analyses. We list a number of attack vectors and related capabilities available to an advanced attacker;
- In Chapter 3, we propose an overview of program analysis techniques, with a focus on symbolic execution and binary analysis. We also explore existing techniques to automatically reason about an attacker able to inject faults into a program;
- In Chapter 4, we detail our formalization of an advanced attacker and the concept of adversarial reachability, as well as an algorithm, adversarial symbolic execution, to verify adversarial reachability in practice;
- In Chapter 5, we present our implementation of adversarial symbolic execution for binary-level analysis inside the BINSEC tool that we call BINSEC/ASE. We propose a user and a developer guide for BINSEC/ASE;
- In Chapter 6, we evaluate BINSEC/ASE performance on standard SWIFI benchmarks and propose few security scenarios to showcase the interest and feasibility of our technique;
- Finally, in Chapter 7 we conclude this thesis and propose future research directions.

[1] https://github.com/binsec/esop2023_artefact
[2] https://zenodo.org/record/7507112

Chapter 2

Context and Motivation

Contents

This chapter aims to motivate the work of this thesis, that is, designing a technique to automatically and efficiently reason about the impact of an advanced attacker on a program's security properties. First, in Section 2.1, we will have a glance at what are security-oriented IT systems, their usages and their properties. They are prevalent in modern life and their attack can have important consequences. Then, in Section 2.2, we will go over different kinds of attack techniques an advanced attacker can leverage against an IT system and what capabilities it gives them over a program.

2.1 Security of Information Technology Systems

In this section, we present IT (Information Technology) systems and their security-oriented features, we stress their prevalence. IT systems are targets for advanced attackers and can have a wide attack surface. We introduce techniques aimed at reducing this attack surface, going toward better system safety and security.

2.1.1 Definition

We call security-oriented IT systems any kind of IT system providing a security feature whose malfunction has bad consequences, knowledge, human, material or monetary-wise for instance.

An IT system is at least composed of a hardware part, the chips themselves, and the software part, the program running on the hardware. It is often combined with input and output systems such as cabled connections, sensors, buttons, antennas, displays or engines, for interactions with the world. Such a system can be on its own as an embedded system like an IoT (Internet of Things) device, or part of a larger IT system like a cloud server.

Many of today's IT systems integrate some form of cryptography to protect the confidentiality of communications, which is one of the most famous security features. This gives an idea of the prevalence of systems with security properties.

2.1.2 Security Properties

In the security field, we typically identify three main security properties a system can have. Knowing they are not always compatible, the strength with which each should be enforced depends on the system application.

- **Integrity:** the system is not to be modified except by legitimate means. In particular, the hardware shouldn't suffer damage, the code shouldn't be altered and the data shouldn't be corrupted. For instance, data is encoded with extra information to detect errors in network transmissions. An example of a malicious integrity breach is the defacement of a website, where an attacker changes the content of a web page against the owner's wishes.

- **Confidentiality:** protected information is to be revealed only to legitimate users. An IT system contains a lot of information, from code to data or even hardware design. Some of that information is labeled as 'public' which means it can be exposed without endangering the confidentiality of the system. Other parts of

the information contained are labeled 'private' and should remain secret. They can be intellectual property like algorithms and hardware design, or secret data such as keys used for encryption or authentication, or personal data. A man-at-the-middle attack consists in intercepting, decoding, observing, re-encoding and sending on its way an information paquet on a network, threatening the confidentiality of those exchanges.

- **Availability:** some systems shouldn't stop working, even in unforeseen conditions like internal errors or when subjected to attack attempts. Availability is measured as a minimum percentage of a system's uptime. Systems integrate recovery mechanisms to restore a working state, fail soft modes or be able to restart the system so it continues its purpose. Denial of service (DoS) attacks target the availability of a system by bombarding it with requests until it cannot handle the number and stops answering requests.

Other types of properties are sought for in security devices. We list some in the following.

- **Authentication:** the use of an IT system's features may be restricted to authorized users only, possibly according to their roles. Each user should register their identity and only perform actions they are allowed to on the system. An unauthorized user is not to forge an authorized user's identity. A brute-force attack can be used to try common passwords until one is accepted and the attacker is allowed entry on a system.

- **Tracability:** each action on a system should be logged in order to be audited. What type of action, on what object and performed by whom should be saved. This is particularly important to detect traces of attackers on a large computer system, determine what has been compromised and where was the exploited breach.

Those are general, system-level security properties. They can be refined into a wide range of other properties at different abstraction levels. For instance, one underlying property of integrity is memory safety. It consists in ensuring memory accesses only affect legitimate objects and does not corrupt or leak other objects.

2.1.3 Wide-spread Use

Many areas of our life use IT systems containing security-oriented features. We provide here a non-exhaustive list of fields and attack examples to stress the need for high security requirements.

Military. The first domain to come to mind may be defense, where IT systems[1,2] have life-and-death consequences on humans. Moreover, the confidentiality and integrity of information, or commands, from the enemy are necessary. There is also a need for preserving intellectual property as a military advantage.

However, a wide variety of fields, in our everyday life, also takes advantage of systems with security-oriented features.

[1] https://www.wired.com/story/dire-possibility-cyberattacks-weapons-systems/

[2] https://www.brookings.edu/techstream/hacked-drones-and-busted-logistics-are-the-cyber-future-of-warfare/

Aeronautics. Airplanes contain many cyber-physical devices to help pilots, anti-collision systems or pressure sensors for instance. They require communication with the airports and control towers. An airplane malfunction can put at risk the lives of hundreds of people. In 2015, a denial-of-service (DoS) attack[3] on the Polish airline company LOT resulted in the cancellation of 10 flights, and 15 more were delayed. The company systems were unable to send flight plans and other information necessary for take off such as weather forecasts. This attack left 1.400 people stranded at Warsaw Chopin airport.

Aerospace. The aerospace field is rich with security-oriented IT systems. They play a big part in communication systems, GPS, weather forecasts and cartography, for civilian and military usage. In addition, humans are sent to space, which is a hostile environment they need to be appropriately shielded from. In 1999[4], a teenager found a flaw in a system of the American Defense Threat Reduction Agency (DTRA) and gained access to messages, including authentication information belonging to NASA. He used it to perform a privilege escalation attack, ultimately downloading the source code of the software running on board the International Space Station (ISS), including temperature and humidity control within living quarters.

Banking. Maybe the first example that comes to mind with security-oriented IT systems present in our pockets is the credit card. It contains private encryption keys to authenticate to a bank and allow payments. Credit card terminals are equally vulnerable. In 2015 and 2016, the SWIFT banking network suffered an attack[5] where attackers, using a malware, gained access to legitimate SWIFT credentials they used to send transfer requests to other banks, who trusted the messages and transferred funds, leading to the theft of more than 100 millions of dollars. Maybe less known are crypto wallets, devices providing encryption keys to access a digital wallet of cryptocurrencies and allow trade. The cryptocurrencies themselves rely on blockchains, and smart contracts where a bug has devastating consequences and is hard to fix. This is a current topic of research [WZ18].

Energy. The power grid is monitored and regulated by a network of devices. It can be attacked to cause blackouts. The first acknowledge successful attack on a power grid targeted Ukraine in 2015[6]. Corporate networks were infiltrated with the BlackEnergy malware using spear-phishing emails. The attack then progressed on to SCADA devices switching them off remotely, and disabling IT infrastructure and emergency power systems, causing a wide power blackout affecting more than 200.000 people. In addition, a DoS attack on the call center kept consumers in the dark.

The water management system is equally vulnerable. In 2021, Florida was the stage of an attack on its water system[7]. An attacker took advantage of a remote access software on a computer not up-to-date with an insufficient firewall rules to try and poison the water supply. Fortunately, it was discovered before any harm could be done to the population.

Entertainment. While the entertainment industry doesn't immediately come to mind when one thinks about security-oriented IT systems, their systems still record per-

[3] https://www.cnbc.com/2015/06/22/hack-attack-leaves-1400-passengers-of-polish-airline-lot-grounded.html

[4] https://cybernews.com/editorial/how-a-florida-teenager-hacked-nasas-source-code/

[5] https://en.wikipedia.org/wiki/2015–2016_SWIFT_banking_hack

[6] https://en.wikipedia.org/wiki/2015_Ukraine_power_grid_hack

[7] https://edition.cnn.com/2021/02/10/us/florida-water-poison-cyber/index.html

sonal data like account and password (sometimes reused elsewhere), but also banking information. In 2012, the PlayStation Network underwent an attack[8] compromising the account of around 77millions of people, including children. Among that personal information was payment data. To limit the leak, the PlayStation network was down for 23 days preventing users from accessing the gaming service.

Health. Another card in our wallet (in France at least) is the social security smart card. More and more health devices, remotely controllable to some extent, are used in medicine to help patients, like pacemakers or insulin pumps. They can be hacked to hurt the people wearing them. In general, hospitals rely on IT systems but are notorious for their lack of cybersecurity investments. For instance, in 2017 in the United Kingdom[9], the NHS was infected by the ransomware WannaCry, blocking access to computers and medical equipment, appointments and operations were canceled and patient information was lost.

Telecommunications. The field of telecommunication includes the Internet and the many routers and relais paving it. They are outside, in the world, and susceptible to disrupting the network or stealing sensitive data. Mobile phone telecommunications can also be targeted, as well as television, radio frequencies, etc. For instance, in 1995, a radio contest in which the 102^{nd} caller would win a Porsche was hijacked[10]. Telephone lines were blocked to make sure only the attacker call could succeed. In 2015, the French television group, TV5Monde fell victim to an attack[11] that targeted the broadcast system. All their channels when black which put the company at risk of losing distribution contracts. Their social media, website and internal systems also went down.

Transport. The rise of autonomous cars [KKJ+21] gives growing responsibilities to manufacturers in traffic accidents and overall road safety. A famous example is a recognition software being derailed by a speed limit sign interpreted as a stop sign. Many other transport systems rely heavily on IT systems, such as boats, trains, traffic control systems, automated subways, drones and UAVs. In 2017, boat navigation was disturbed[12] by an attack on the GPS system, placing some boats inland.

2.1.4 Attack Surface and Attackers

Security-oriented IT systems are a prime target for attackers, who would exploit vulnerabilities from any component and at any stage of the product life-cycle.

2.1.4.1 A Wide Attack Surface

IT systems are complex systems, embedding many different layers which can all have vulnerabilities: the hardware, the micro-architecture, software-controlled hardware mechanisms, kernels, operating systems, network, software, algorithms or protocols. IT systems can reside in physically secure server rooms, providing their services through

[8] https://en.wikipedia.org/wiki/2011_PlayStation_Network_outage

[9] https://www.nao.org.uk/reports/investigation-wannacry-cyber-attack-and-the-nhs/

[10] https://www.industrialcybersecuritypulse.com/threats-vulnerabilities/throwback-attack-kevin-poulsen-wins-a-porsche-and-hacks-the-u-s-government/

[11] https://en.wikipedia.org/wiki/TV5Monde#April_2015_Cyberattack_and_resulting_disruption

[12] https://www.newscientist.com/article/2143499-ships-fooled-in-gps-spoofing-attack-suggest-russian-cyberweapon/

a cabled connection, usually over the Internet. As getting proximity access to such devices is difficult, attackers favor software-based attacks. Security-oriented IT systems can also be embedded devices, that is to say, small systems scattered in the world where they are needed for their features. Since embedded devices are in the open, an attacker can have direct contact with the hardware. It enables them to perform, in addition to the wide spectrum of attacks possible on physically secure systems, hardware fault injection attacks. We detail different types of attack vectors and what capabilities they provide an attacker in section 2.2.

Some but not all attack vectors rely on existing *vulnerabilities, or bugs*, in an IT system. A vulnerability is a weakness in a system that can be exploited. It can be a design flaw or a development error for instance.

2.1.4.2 Attack Surface Through Time

In addition to being complex systems, IT systems also have a rich life cycle, each step possibly introducing vulnerabilities. The Common Criteria [CC2] detail different moments in the life-cycle of a product where vulnerabilities may arise through failures in IT:

- *requirements*: an IT product may possess all the functions and features required of it and still contain vulnerabilities that render it unsuitable or ineffective with respect to security;
- *design*: an IT product has been poorly designed. Building a secure product, system, or application requires not only the implementation of functional requirements but also an architecture that allows for the effective enforcement of specific security properties the product, system, or application is supposed to enforce. The ability to withstand attacks the product, system, or application may be face in its intended operational environment is highly dependent on an architecture that prohibits those attacks or, if they cannot be prohibited, allows for detection of such attacks and/or limitation of the damage such an attack can cause;
- *development*: an IT product does not meet its specifications and/or vulnerabilities have been introduced as a result of poor development standards or incorrect design choices;
- *delivery, installation and configuration*: an IT product has vulnerabilities introduced during the delivery, installation and configuration of the product;
- *operation*: an IT product has been constructed correctly to a correct specification, but vulnerabilities have been introduced as a result of inadequate controls upon the operation.
- *maintenance*: an IT product is maintained in such a way that new vulnerabilities are introduced.

2.1.4.3 Protections Adding Attack Surface

Interestingly, removing vulnerabilities in a device by adding protections can lead to adding components, micro-architectural or software, in the systems, which can themselves suffer from vulnerabilities or induce new ones in other parts. Hence, adding countermeasures also increases the attack surface of a system.

This stresses the high potential for a wide range of vulnerabilities to exist in a security product.

2.1.4.4 Malicious Attackers

There is a wide diversity of malicious attackers with various motivations from monetary to politics. The most serious threats to security-oriented systems are powerful entities such as mafias and state agencies. They have enormous resources in terms of time, money to buy equipment and in terms of expertise. It is reasonable to assume such attackers will try every attack vector at their disposal and target any stage of the product life-cycle. They may even use some new vulnerabilities or attack techniques not yet known to the public. Those are called zero-day vulnerabilities [BD12]. They are hard to detect and impossible to protect against before they are publicly disclosed. Zero-day vulnerabilities can exist for years and are traded between malicious parties.

Malicious attackers pose a real threat to security-oriented IT systems. Techniques and processes have been developed to harden system security and add considerable difficulties for malicious attackers.

2.1.5 Ensure Security

To harden security-oriented IT systems against attackers, different *complementary* approaches have been developed.

2.1.5.1 Standard System Protections

The first step to producing a secure product is to produce a 'good' product. Developers exchange good practices for efficient but also readable and self-explanatory code and specifications. Companies put in place development processes like code review to reduce the risk of bugs.

The design choice can have a big impact on reducing the risk and number of vulnerabilities. For instance, the choice of programming language can prevent some classes of bugs. A type-safe language contributes to program correctness and prevents type errors like using operations with incorrect data types. A memory-safe language enforces memory isolation.

When the chosen programming language is compiled, a generic tool is used, GCC[13] or Clang[14] to compile C for instance. They integrate by default a number of security protections, like stack canaries, and can be configured more precisely.

Limits. Those standard security practices permit the removal of some vulnerabilities, but are mostly human-based, hence fallible, or take advantage of default configurations that are rather weak and not tailored to a particular system.

2.1.5.2 Product Validation

To complement standard development processes, various test phases have been designed to improve functional correctness and detect bugs early. Unit tests are written or generated, for each functional unit, verifying correct output for a limited number of inputs. This help make sure of the normal behavior of the system, i.e. functional

[13] https://gcc.gnu.org/
[14] https://clang.llvm.org/

correctness. Then, integration tests check that each component interacts with others as expected, as many imprecisions or bugs lay at system interfaces, which can then be exploited by attackers. Regression testing ensures new developments haven't impacted the nominal behaviors of the other parts of the system.

Limits. This validation procedure allows to remove most vulnerabilities in a system. However, one of the main difficulties with tests is the coverage of all functional behaviors. If a set of paths is not explored by a test, its correctness isn't checked. Exhaustiveness is a hard problem and can result in very heavy test suites, too heavy for some development processes. Incomplete coverage often happens for corner cases, that an attacker can exploit. For instance, what happens if an element of a data structure is empty? In addition, test cases are often selected with functional testing in mind, that is to say, what the system should do, and not security, i.e. what the system shouldn't do. In particular, incorrect inputs are not always exhaustively tested.

2.1.5.3 Security Evaluation

The next step in strengthening an IT system is to perform *penetration testing*. It is a security-oriented testing phase to *evaluate the security by design in practice*. Penetration testing consists, for security practitioners, in attacking the system as a malicious attacker would to identify the remaining vulnerabilities and available attack vectors. Different entities perform penetration testing on IT systems.

Manufacturer. It is the entity designing new IT systems they wish to market. They may externalize the hardware manufacturing and even the software development. However, they are responsible for the end product security. Manufacturers test their products and look for vulnerabilities to remove them as early as possible in the production process as making patches becomes more and more expensive. This includes trying to attack their product, to ensure a certain level of robustness to their clients.

Evaluators. Penetration testing can be externalized to evaluators. They can work for manufacturers or certification authorities (see Section 2.1.5.4). Evaluators receive an IT system, perform its vulnerability assessment including penetration testing, and produce a report.

Security Researcher. Researchers study the security of IT systems to improve the state of knowledge, discover new types of vulnerabilities, develop new tools to find them and devise new and more powerful countermeasures. Their work empowers the security community, helping improve their assessments. It is likely similar to the research made by malicious attackers, in particular, to discover zero-day vulnerabilities in systems.

2.1.5.4 Certification Process

Each company has its own development process, making it hard to compare systems quality and security. To bridge that gap, independent authorities receive product applications and attribute to them a standardized label. Those are *certification authorities*, delivering *certifications*.

Certification Authorities. At an international level, countries have gathered around Common Criteria. It takes the form of a procedure to obtain the Common Criteria certificate, with different security levels. Other certifications exist, at the national level for instance, in France, the CSPN can be delivered by the French National Cybersecurity Agency (Agence Nationale de la Sécurité de Systèmes d'Information, or ANSSI

for short).

Certification. The application consists of extensive documentation of the product, development and test processes followed by the company. It also integrates a penetration testing report from an external evaluation center. For instance, among the documentation required for the Common Criteria certification, the class AVA relates to vulnerability assessment, where the evaluator is expected to report on penetration testing activities to ensure the non-exploitability of the product's vulnerabilities [CC2]. The certification authority reviews the application and evaluates the product made available to them to decide whether or not to give the certification to the new product.

2.1.5.5 Our Scope

The work presented in this thesis focuses on *program-level evaluation for security*. It aims to identify vulnerabilities in an IT system that an advanced attacker could exploit. It could assist penetration testers identify sensitive code areas and possible attack paths. We propose for instance the following use cases:

- the evaluation of the strength of different cryptographic implementations;
- the evaluation of security features of security-oriented IT systems such as a secure bootloader;
- the vulnerability evaluation of a program with various countermeasures.

2.1.6 Example of Attack on a Security System

As a motivating example, we consider a pin verification program. It can be used for different applications, for instance, for credit cards, building access control, secure storage devices, or authentication of any kind. Here, an attacker may want to leak the secret pin, or they can confuse the system into believing the attacker has the secret pin even if not, in order to continue and use authenticated features. More formally, the attacker's goal is to be authenticated despite not knowing the correct pin.

The first step for an attacker would be to retrieve the program, either from public sources or by leaking it. The motivating example is illustrated in Figure 2.1, it is an unprotected version of a VerifyPIN program [DPP+16], from the domain of hardware fault injection and smart cards. The user pin digits, stored in $g\_userPin$, are checked against the reference digits stored in $g\_cardPin$, using the $byteArrayCompare$ function. The attacker seeks to be authenticated (satisfy the assert l.29) without knowing, i.e. entering, the right digits.

We propose hereafter different attack scenarios for the attacker to exploit the VerifyPIN program.

Attack Scenario 1. Once the attacker has access to the program code, they would analyze it for known *Common Vulnerabilities and Exposures* (CVEs) and classic *exploitable vulnerabilities*, checking for a buffer-overflow vulnerability for instance.

Attack Scenario 2. During their analysis of the program, the attacker can notice this code is not constant-time. That is to say, a program branch depends on a secret value (l.10) leading to different execution times and information leakage about the secret. Here the program will take longer to execute, i.e. execute one more loop iteration,

```
1 #define PIN_SIZE 4
2
3 bool g_authenticated = 0;
4 int g_userPin[PIN_SIZE];
5 int g_cardPin[PIN_SIZE];
6
7 bool byteArrayCompare(int* a1, int* a2, int size) {
8     int i;
9     for(i = 0; i < size; i++) {
10        if(a1[i] != a2[i]) {
11            return 0;
12        }
13    }
14    return 1;
15 }
16
17 void verifyPIN() {
18    g_authenticated = 0;
19    if(byteArrayCompare(g_userPin, g_cardPin,
20        PIN_SIZE) == 1)
21    {
22        g_authenticated = 1; // Authentication();
23    }
24    return;
25 }
26
27 void main() {
28    verifyPIN();
29    assert(g_authenticated == 1);
30 }
```

Figure 2.1: Motivating example, inspired by VerifyPIN [DPP+16]

for each correct digit. By performing a timing side-channel attack, the attacker can leak the secret pin by learning one digit at a time, testing all 10 possibilities. This would require at most $10 \times PIN\_SIZE$ attempts which is much lower than brute force (10^{PIN_SIZE} attempts). Then the attacker can use this information to get legitimately authenticated by the program.

Attack Scenario 3. If they come up empty-handed, attackers could try and determine sensitive code areas, that is to say, code regions where inducing a fault in the execution is likely to result in a different program behavior, closer to what the attacker aims for. More details on means to inject faults into a program are presented in Section 2.2. If the injection of the fault requires precise timing, for *hardware fault injections* or software-induced hardware fault injections for instance, the attacker would need to observe the running program, through some form of side-channel leakages like power consumption or electromagnetic emission, to link processor activity with program execution in order to synchronize the injection with the execution of the vulnerable area. The attacker may need multiple attempts, changing the injection vector or the vulnerable code region

before performing a successful injection.

Attack Scenario 3A. The attacker could use fault injection to change the *control flow* of the program, by inverting the test l.19 which checks the return value of the *byteArrayCompare* function, so that when it returns 0 (pin incorrect), the execution would in fact execute l.22 and authenticate the user.

Attack Scenario 3B. The attacker could target *data in memory*. In this example, injecting a non-controlled fault into memory to corrupt *g_cardPin* would not help the attacker. If they can inject a precise fault, resetting the whole memory array associated with *g_cardPin*, then entering a "0 0 0 0" pin would authenticate them.

Attack Scenario 3C. The attacker can target the *data flow* of the program. By corrupting the initialization of *g_authenticated* l.18 with a fault injection setting it to 1, the program will detect the pin entered is wrong and forward the original value of *g_authenicated*. Hence the attacker will be authenticated with an incorrect pin.

2.1.7 Conclusion

In this section, we had a look at various fields relying heavily on IT systems with security features and what efforts were put in place to harden their security against attackers, for instance through a Common Criteria certification. The VerifyPIN motivating example illustrated our focus on program-level security evaluation.

Cyber-security is a cat-and-mouse game, where attackers and security researchers develop new attack techniques while manufacturers and other security researchers try to detect them and design countermeasures. Hence protections are imperfect and still leave ways for an advanced attacker to build successful attacks. In the next section, we will delve into what attack techniques are available to an advanced attacker and what attack capabilities they provide the attacker to reach their goal.

2.2 Capabilities of a Powerful Attacker

Security research has delved into many aspects of system security, leveraging a growing variety of attack techniques and attack surfaces that a powerful attacker can take advantage of.

We aim in this section to touch on what impact an advanced attacker can have on a program. We consider many attack vectors from software to micro-architecture and hardware. We list the attack capability given to an attacker by each kind of attack. It is by knowing against what to defend a program that it is possible to evaluate its security and strengthen it.

2.2.1 Side Channel Attacks

A system running a program leaks a lot of information about its internal state. It can be through external observables like power consumption [KJJ99], electromagnetic emissions [AARR03, DOL+10], execution time [Koc96] or light emission [SNK+12], and more recently through sound emissions [GST17]. An attacker able to run a program on the target processor can also observe internal leakages like cache usage [Ber05, LM18] or branch prediction [AKS06]. An attacker observing those leaks deduces private information, like interesting addresses or secret keys. As the attacker does not modify

the system's behavior, it is a *passive attack*. We will not detail side channel attacks further and focus on active attacks in this thesis.

2.2.2 Software Attacks

Many tools have been developed to ensure system's security, mostly taking into account an attacker able to craft malicious inputs to give to legitimate input sources of a program. Software attacks exploit existing software vulnerabilities. A number of them are recorded in the CVE database[15]. They can be performed remotely. We list below the most common ones and the attack primitive they give to an attacker.

Buffer Overflows. It is probably the most famous one. This vulnerability appears when the program allows to write in memory outside the bound of a data structure, arrays in particular. This is an undefined behavior in the C language that is usually resolved by allowing the write outside of bounds. This vulnerability is often due to a code pattern where a write operation is performed inside a loop and there end up being too many iterations of the loop. It can also appear when copying a data structure into a smaller one. Figure 2.2, from OWASP[16], illustrates a code relying on external data to control its behavior. The code uses the *gets*() function to read an arbitrary amount of data into a stack buffer. Because there is no way to limit the amount of data read by this function, the safety of the code depends on the user to always enter fewer than $BUFSIZE$ characters.

```
1 char buf[BUFSIZE];
2 gets(buf);
```

Figure 2.2: Example of a buffer overflow vulnerability.

A buffer overflow attack [CWP+00] gives the attacker write capabilities. The specificities of the code will constrain what can be overwritten and how much of it. A common usage is to overwrite the return pointer at the end of a function and start a return-oriented type of attack.

Use-after-free. This type of attack [CGMN12] consists in dereferencing a pointer after it has been released by the code. An attacker can use that pointer to perform illegal memory accesses, on data or function pointers, retrieving old values or new ones allocated after the space had been freed. A use-after-free attack enables an attacker to control values from memory loads and jump to attacker control code.

Integer overflow. Integer overflow [DLRA15] consists in providing inputs such that an arithmetical computation will overflow –or underflow, that is to say, the theoretical result cannot be written with the chosen integer representation. This is an undefined behavior in the C language. This vulnerability can be used to corrupt data and even control flow when targeting loop iterations.

Conclusion. Software vulnerabilities provide the attacker with a number of interesting attack primitives, mostly based on data corruption with extended effects. Table 2.1 summarizes attacker capabilities. Here we only consider the impact of exploiting such a vulnerability and presume attack prerequisites have been met. Interestingly, works on

[15] https://cve.mitre.org/
[16] https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

Control Flow Integrity (CFI) mechanisms [ABEL09, BCN+17] also define hypothetical attackers by their capabilities, such as "write anything anywhere" or "write anything somewhere", in order to prove that their countermeasure is indeed able to thwart such an opponent.

Table 2.1: Attacker capabilities given by software attacks

Type of attack	Attacker capability
Buffer overflow	Overwrite data in memory
Use-after-free	Corrupt data loads from memory
Integer overflow	Corrupt input-related data

2.2.3 Hardware Fault Injection Attacks

When an attacker has direct access to a device, they can put it in extreme environmental conditions, past the operating limits of the components, to induce hardware faults in them. The faults occur at the circuit level, either setting a signal to zero (bit reset), to one (bit set) or flipping the value (bit-flip). The consequences of that circuit-level fault will manifest at the micro-architectural level depending on the location of the faults, for instance at the fetch, decode, execute or store stage of the microprocessor pipeline, in registers, cache or memory, etc. Finally, faults propagate to the software layer, changing the behavior of the executed program. Faults can be *transient*, with effects lasting only until they are naturally overwritten, or *permanent*, when they cause lasting damage to the circuit. We will only consider transient fault in this thesis. Security practitioners have identified various injection means [BECN+06, KSV13, BBKN12]. We list the most common ones in the following, with what types of capabilities they are known to provide an attacker.

Laser Beam. A photoelectric effect happens when a material is hit by electromagnetic radiation such as a laser beam, the material releases electrons creating electron holes that induce currents and cause faults [DDCS+14]. This is the most accurate hardware fault injection mean, with a timing precision of a few picoseconds [LBC+15] at a location precise at a few micrometers [BMPM13]. A laser beam injection can affect each stage of the pipeline (instruction fetch, decode, execute, write) [VTM+17]; for instance corrupting operands, corrupting instructions or writing wrong results back in a possibly different register. A laser beam can also inject a bit-flip in a register, a flag value [VTM+17] or in flash memory [CMD+19] independently of program execution. A concrete attack could be flipping a precise bit in the secure configuration register (SCR) to bypass a secure boot [VTM+17]. A laser beam can inject multiple faults in one attack:

- at different physical locations [CGV+22], for instance, precisely faulting each check of a pin verification code [DRPR19];
- affecting consecutive instructions [RNR+15]. Interestingly, if we consider an attacker capable of an infinite number of precise skips, they have a Turing complete computing power [PCHR20]. In practice, up to 300 instructions have been skipped [DRPR19].

Power glitch. The critical path is the signal taking the longest to get from one flip-flop or memory unit, through some logic gates and to the next flip-flop or memory unit. Under-powering a system will slow down signals until some signal in the critical

path cannot reach the next sampling unit in time. Those are called *timing faults*. Power glitching allows for attacks with precise timing but no spatial control. The power supply of a component must be accessible. A power glitch can skip instructions or influence a branch decision, corrupt memory locations, or alter the result of an instruction or its side effects [BFP19]. Power glitches can be used for privilege escalation on a Linux system [TM17]. Multiple power glitching faults are possible [BFP19] and can be leveraged to extract a protected firmware with more than a hundred faults to force memory accesses.

Clock glitch. The clock gives the pulse at which rate flip-flops and memory units sample signals. When the frequency is increased passed the maximal frequency safe for the critical path, timing faults occur. The location of a clock glitch is architecture-dependent but precise in timing. The device must have an external clock that can be changed to an attacker-controlled one. A clock glitch in the CPU pipeline can corrupt instructions by affecting an operand fetch or skipping an instruction [YGS+16], with or without replaying the previous one [CPHR21]. A glitch injection platform [CPHR21] has skipped a hundred consecutive instructions and performed multiple bursts of skips. One clock glitch can bypass unprotected authentication and multiple faults, a protected one [CPHR21].

Electromagnetic Pulse (EMP). The electromagnetic field of an EM probe generates by induction eddy currents in the chip. This also creates timing faults in the circuit [ZDT+14, OGSM15]. EMP attacks are temporally and spatially precise [MDP+20]. With a low-intensity electromagnetic disturbance, instructions can be corrupted to other instructions, and at high-intensity, instruction's op-code are zeroed [BGV11]. EMP has been shown [MDH+13] to inject faults either on the instruction bus, for instance targeting branch instructions, ALU instructions or load-store instructions, and on the data bus independently of the execution, bringing each faulted bit closer to the precharge bit value (architecture dependent). Multiple bit-flips or most-significant half-word reset on register values, operand substitutions and instruction skips or replay were also observed [PHB+19]. EMP can induce vulnerabilities such as a buffer overflow and enables control flow hijacking, covert backdoor insertion and Return Oriented Programming [BLLL18]. Multiple consecutive instructions can be corrupted [RNR+15].

Conclusion. Hardware fault injection attacks have various effects, usually described as modifications of instructions, or data corruption. We list the most common ones in Table 2.2. In addition, *multiple faults* can be injected with a hardware attack. Capabilities provided by hardware fault injection are the building blocks of exploitation:

- breaking cryptography [BS97, Gir05, BSGD09, BBB+10, BBKN12, VKS11, DDRT12, HS14],
- leaking sensitive data [BDL01, DMM+13, BFP19],
- circumventing authentication [QS02, DPP+16, DRPR19, CPHR21],
- bypassing software protections [NHH+17, BLLL18], sometimes in parallel with a software attack exploiting the created vulnerability as a *combined attack* [TS16, NHH+17].
- bypassing secure boot [TS16, VTM+17, BFP19],
- performing privilege escalations [TS16, TM17];
- disrupting a system whose output is studied by side-channel analysis [AVFM07].

Table 2.2: Attacker capabilities given by hardware fault injection attacks

Type of attack	Attacker capability
Hardware fault injection attack: laser beam, power glitch, clock glitch, temperature, EMP	Skip an instruction Skip and replay the previous instruction Change operator of an instruction Change source register of an instruction Corrupt load from memory Change destination register of an instruction Corrupt a store instruction's value Corrupt a memory cell Corrupt a register Corrupt a flag

2.2.4 Software-Implemented Hardware Attacks

We distinguish software-implemented hardware attacks from traditional hardware attacks due to the source of the perturbation. While hardware perturbations are caused by extreme environmental conditions (external) in hardware attacks, they are caused by software-controlled mechanisms (internal) in software-implemented hardware attacks. The goal is to push the hardware into unstable states using legitimate software commands. While hardware-controlled injections require proximity to the target, software-controlled injections can be done remotely. We list some examples of software-controlled mechanisms that can be misused to inject faults into a program and their associated fault models.

Rowhammer. In 2014, Kim [KDK+14] first discovered that high-intensity memory access to the same location in memory can induce a bit-flip in the adjacent memory row, in DRAM (Dynamic Random Access Memory). The trend of circuit miniaturization leads to memory cells closer to one another and their charges being able to disturb each other. This was named the Rowhammer vulnerability. It was then exploited in 2015 to escape a sandbox and to perform a privilege escalation [SD15] by corrupting the page table entry (PTE) to point to the attacker's one and control all physical memory, hence the entire system. Since then, a number of works have developed other exploits, on many different platforms [MK19]. It provides the attacker with a bit-flip capability at a precise location anywhere in flash memory [RGB+16].

Memory Delays. The time a signal takes to travel between the memory and the CPU mostly depends on the length of the wire connecting them. However, that delay can also change with voltage and temperature. To accommodate for the delays changing from one board to another and due to working conditions, hardware timing control mechanisms have been implemented, called delay-lines. They are usually configured at boot but remain programmable. After being used for power side-channel attacks [GDTM21b], exploiting the link between voltage fluctuations and memory delays, this software-controlled mechanism was turned into a fault injection mean, named FaultLine, by Gravellier *et al.* [GDTM21a]. By modifying the delay-line to a shorter period, an attacker can induce faults in memory transfers (loads and stores) of a victim application. This provides the attacker with the capability to induce a precise number of bit-flips either in memory loads or memory writes, or both. For instance, it allowed Gravellier *et al.* to perform a BellCore attack on OpenSSL signatures.

Dynamic Voltage and Frequency Scaling (DVFS). Hardware manufacturers have developed CPU with energy management controls, adapting frequency and voltage to increase performance in high-intensity computation phases and reduce consumption in low-intensity phases. Those configurations are stored in registers that can be controlled by kernel drivers, leading to potential misuse. Tang *et al.* devised the first exploit technique, CLKSCREW, based on the DVFS vulnerability in 2017 [TSS17]. They show the microprocessor can be put in an unstable state, with the same consequences as under-powering or overclocking. The attack is imprecise in location and induces faults in flip-flops. DVFS has then been shown in the Plundervolt attack [MOG+20] to induce memory safety vulnerabilities and recover secret keys from cryptographic programs on an SGX enclave.

Conclusion. As systems become more complex, they often require more modularity to adapt to different boards and functioning conditions. Hence a number of software-implemented hardware mechanisms to keep signal synchronization and improve energy efficiency. However, they often remain programmable hence vectors of attacks.

Software-implemented hardware attacks can yield the same types of faults than hardware fault injection, summarized in Table 2.3, while lifting the direct access restriction. They can be used for the same classes of attacks, from memory corruption to privilege escalation and cryptography attacks.

Table 2.3: Attacker capabilities given by software-implemented fault injection attacks

Type of attack	Attacker capability
Rowhammer	flip a bit in memory
Faultline	bit-flips in memory transfers
DVFS	flip a bit somewhere

2.2.5 Micro-Architectural Attacks

Manufacturers have design a wide variety of acceleration mechanisms to improve the performance of their CPUs in parallel with the miniaturisation trend. Those mechanisms can be misused to yield attacks. We present some of them in this section.

Spectre Attack. To improve modern processors' performance and, in particular, to avoid having to wait for a conditional branching resolution, branch prediction and speculative execution have been implemented. Branch prediction consists, for the processor, in making a guess as to which branch will be taken, based on previous decisions. For instance, if the last hundred of time, the branch true was taken for a particular conditional jump, the branch predictor can guess the true branch will be taken again this time. The register state is saved and the processor enters a speculative execution mode, where it continues executing the guessed branch instructions instead of doing nothing, waiting for the branch result. Most of the time, the branch predictor is correct and the processor gains time. Sometimes it is wrong, the processor state is restored and the execution starts again on the other branch, which is not slower than simply waiting for the branch result.

The Spectre attack [KHF+20] trains the branch predictor to its advantage to exploit misprediction, i.e. when the branch predictor was wrong and started executing instructions it shouldn't have, in addition to the observable effects left after the roll-back, like

cache content. In effect, the attacker is able to invert a test during the speculative window. Many variants exist. Variant 1 allows an attacker to read from arbitrary memory, variant 2 corrupts indirect branch target computation to deviate the execution to an arbitrary location.

Load Value Injection (LVI). This attack [VBMS+20] also exploits speculation execution, but from the data perspective. When a load operation fails due to a page fault, the processor can transiently use data from a micro-architectural buffer to increase execution speed. The attacker controlling the buffer can inject arbitrary data from any load operation, including implicit load micro-op like the x86 *ret* instruction.

Race attacks. Modern CPU cores can run multiple processes at once, interleaving their execution with a scheduler. If those processes share some part of memory, they can interfere in the other's data flow [GSV03]. For instance, in a Time-Of-Check-To-Time-Of-Use (TOCTOU) attack [WP05], the victim program will first sanitize some user input, then the malicious program will overwrite said data with malicious and malformed content, which will be processed by the victim as if correctly formed, causing unintended behavior the attacker can exploit. Race attacks give write capabilities to the attacker, on data in memory which includes saved register states.

Conclusion. To increase the execution speed of processors, micro-architectural behaviors have been developed like scheduling or speculative execution. Unfortunately, those added behaviors also increase the attack surface of the target and offer new possibilities for the attacker to induce unintended behavior. A summary of the capabilities due to micro-architectural attacks is presented in Table 2.4.

Table 2.4: Attacker capabilities given by micro-architectural attacks

Type of attack	Attacker capability
Spectre	Invert a test (transiently) Read from arbitrary memory (transiently) Jump to arbitrary location (transiently)
LVI	Write data load results, from any, even implicit load (transiently)
Race attack	Write data in memory and registers

2.2.6 Man-At-The-End (MATE) Attacks

In the context of a MATE attack, the attacker has unlimited access to the target, meaning full observability and control over a software code and its execution [ASA+15]. The attacker can look at what is executed, inspect each instruction and possibly modify it or any architectural element (register, memory) before executing it. For the victim, considering MATE attacks is equivalent to executing code on an untrusted environment. The goal of a MATE attack is usually to steal intellectual property such as sensitive data or code through reverse engineering attacks. This can be done with direct access to the device or remotely. Another type of MATE attacker is a malicious trusted user, an unhappy system administrator for instance. They can use their knowledge of a system and through legitimate access corrupt or erase data, or crash services, machines or an entire IT system. A MATE attacker is also in a unique position to observe protections and monitoring systems in place. Bypassing them is part of the attack. They can for instance redirect a hardware interrupt signal triggered by their attack actions.

The associated attacker model is hence very powerful, with capabilities such as halting the execution, modifying data in memory and registers, and modifying instructions, at any point of the execution.

2.2.7 Summary

We saw in this section that a wide variety of attacks vectors are available to an advanced attacker. Those attacks can be performed multiple time in one attack and in combination. We described several real software-level security scenarios where the attacker goes beyond crafting legitimate input to abuse the system under attack. The effects of the attacks described can be represented using fault models.

2.3 Conclusion

Security-oriented IT systems are omnipresent. They need to be able to resist powerful attackers able to exploit various kinds of attack techniques and attack surfaces. We presented software attacks exploiting programs' inputs, hardware fault injection attacks attacking the physical layer, software-implemented hardware attacks targeting software-controlled hardware mechanisms, micro-architectural attacks misusing micro-architecture behaviors, and MATE attacks with full control over the program and its execution.

It is important to be able to describe in a generic way the capabilities given to an attacker by all those attack vectors. We do this by using fault models, inspired by the hardware fault injection field. It allows to reason about attacks in an abstract way, focusing on the impact an attacker can have on a program and evaluate a program's resistance. In particular, a unified description of attacker capabilities is easier to integrate into generic tools to assist security practitioners in their program-level security assessment.

To evaluate the vulnerability of programs to such an attacker model, from a program hardening or vulnerability detection point of view, code analysis techniques exist as detailed in the next chapter, Chapter 3.

Chapter 3

Background

Contents

We saw in Chapter 2 a program can include security properties that should withstand a certification procedure, its production environment and attackers. In this background chapter, we introduce program analysis techniques used to verify programs in Section 3.1, with a stronger focus on symbolic execution (Section 3.2) and binary analysis (Section 3.3) that the work described in this manuscript builds upon. In addition, we present in Section 3.4 the general topic of software-implemented fault injection, which consists of tools to evaluate program properties in the presence of an attacker able to inject faults.

3.1 Program Analysis Techniques

In this section, we introduce the concept of automated program analysis and present the main families of analyzer properties such as under- and over-approximation, and correctness and completeness. Lastly, we present the notion of reachability which we will build upon in this thesis.

3.1.1 Overview

First, we provide an overview of why program analysis techniques are needed to prevent bugs in programs and present some general program properties that can be verified.

3.1.1.1 Bugs in Programs

Writing code without vulnerabilities or bugs is a hard problem, especially with systems ever growing in complexity. Techniques such as good coding practices or code review and manual testing help in the matter but do not provide strong guarantees. For instance, the OSS-Fuzz platform performs continuous fuzzing on over 300 open-source projects. It has discovered 23,907 bugs in 4 years [DLG21]. There is a need for program analysis techniques able to detect vulnerabilities in programs.

3.1.1.2 Program Analysis

To analyze a program is to determine if the *program behaviors satisfy a property*. The terms 'program behavior' and 'property' can mean different things. Here, we consider a program behavior to be the sequence of states it goes through following an execution path, and a property to be an oracle over the program behaviors. Properties are mainly divided into two classes, safety and liveness.

Safety. A safety property prescribes that *something bad will never happen*. Some examples are:

- There is no division by 0;
- The program never crashes;
- *Memory safety*: no illegitimate, out-of-bounds memory accesses are performed;
- The program does not deviate from specifications.

Liveness. A liveness property prescribes that *something good will eventually happen* in the program execution. Here are a few examples:

- *Availability*: data provided to a program will eventually be treated;
- *Termination*: a program will eventually terminate.

Formalization. More formally, a program $\mathbb{P}$ satisfy a property $\mathcal{P}$, written $\mathbb{P} \models \mathcal{P}$ if and only if $\mathbb{P} \subseteq \mathcal{P}$. That is to say, all behaviors of $\mathbb{P}$ are in the set $\mathcal{P}$ of all possible program behaviors satisfying the property. Conversely, a program $\mathbb{P}$ violates a property $\mathcal{P}$, written $\mathbb{P} \not\models \mathcal{P}$, if and only if $\exists c \in \mathbb{P}.\ c \notin \mathcal{P}$. A program behavior that violates a property is called a *vulnerability* or a *bug*.

Formal methods [CW96] are mathematically based techniques developed to assess properties on programs, or find counter-examples, i.e. bugs.

3.1.2 Program Analysis is Undecidable

Unfortunately, Rice theorem [Ric53] states that the task of determining if a program satisfies a non-trivial property is undecidable. A program analysis cannot be at the same time precise, automatic, terminating and generic in supported programs. In practice, program analysis techniques relax one or more of those attributes to get interesting results.

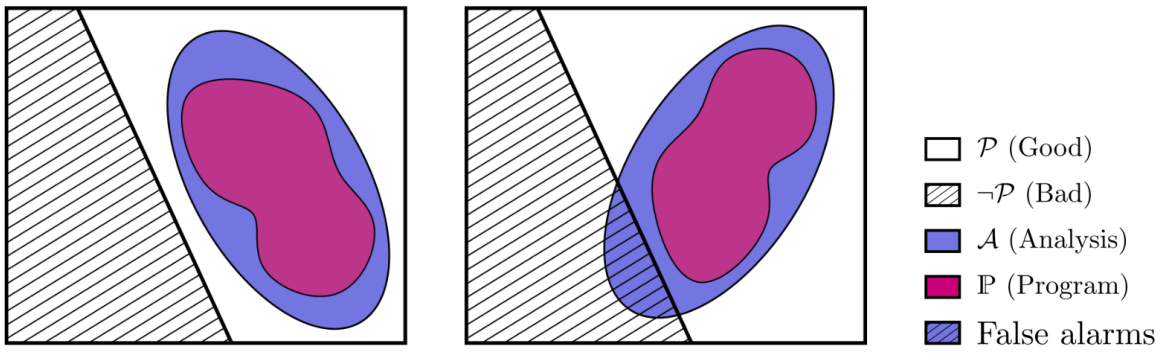
For instance, deductive verification [HH19] is not automatic. It can require manual annotations such as pre-conditions, post-conditions, invariants, or to fully write proofs in some techniques.

A number of program analysis techniques relax precision to obtain results. We explore hereafter the two main categories, under-approximation and over-approximations.

3.1.2.1 Over-Approximation

An analysis over-approximates all possible program behaviors $\mathbb{P}$ by $\mathcal{A}$, a set of computed behaviors that include $\mathbb{P}$ ($\mathbb{P} \subseteq \mathcal{A}$), as illustrated in Figure 3.1. It provides a strong guarantee that the program satisfies the property when the analysis proves the over-approximated behavior satisfies the property. In particular, it can prove the absence of bugs.

$$\mathbb{P} \subseteq \mathcal{A} \wedge \mathcal{A} \subseteq \mathcal{P} \Rightarrow \mathbb{P} \subseteq \mathcal{P} \Rightarrow \mathbb{P} \models \mathcal{P}$$



(A) Example of *sound* analysis: the analysis successfully shows that the program is secure.

(B) Example of a *false alarm*: the analysis reports a bug while the program is secure.

Figure 3.1: Illustration [Dan21] of an analysis $\mathcal{A}$ that over-approximates the behaviors of a program $\mathbb{P}$

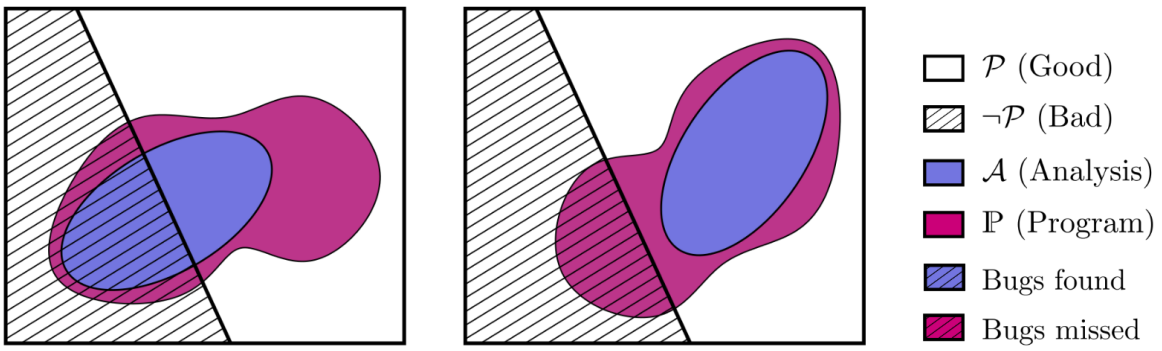
Because the analysis over-approximates program's behaviors, when a bug is found, it may be a real one belonging to actual program behavior, or it can be a false positive, i.e. it is not part of program behavior.

The notion of over-approximation relates to completeness. A program analysis technique is said *complete* when all program behaviors are studied. In particular, the analysis will find all violations of $\mathcal{P}$. An over-approximation-based technique is usually complete but not correct.

An example of a program analysis technique based on over-approximation is *abstract interpretation* [Cou21, RY20]. The set of possible values for an object is over-approximated by a superset with interesting mathematical properties. A well-known example is representing possible integer values of a variable by an integer interval.

3.1.2.2 Under-Approximation

Another way to relax precision in an analysis is to under-approximate program behaviors, i.e. by missing some of them. The set of computed program behaviors $\mathcal{A}$ is included in the actual program behaviors $\mathbb{P}$ ($\mathcal{A} \subseteq \mathbb{P}$) as illustrated in Figure 3.2. Here, if a bug is found by the analysis, it corresponds to a true problematic program behavior.



(A) Example of *bug-finding*: the analysis successfully shows that the program is insecure.

(B) Example of *unsound* analysis: the analysis fails to find bugs while the program is insecure.

Figure 3.2: Illustration [Dan21] of an analysis $\mathcal{A}$ that under-approximates the behaviors of a program $\mathbb{P}$

$$\mathcal{A} \subseteq \mathbb{P} \wedge \exists c \in \mathcal{A}.\ c \notin \mathcal{P} \Rightarrow \exists c \in \mathbb{P}.\ c \notin \mathcal{P} \Rightarrow \mathbb{P} \not\vDash \mathcal{P}$$

A program analysis technique based on under-approximation will not have false positives but can miss bugs since some program behaviors elude it. Hence, such an analysis technique cannot prove the absence of bugs in a program.

The notion of under-approximation is related to the correctness of an analysis. A program analysis is said *correct* when all bugs found are actual program behaviors. An under-approximation based technique is typically correct but not complete.

There are some common program analysis techniques based on under-approximation.

- An example of under-approximating program analysis technique is *fuzzing* [God20]. Fuzzing consists in generating a large number of program inputs and detecting when the execution of the analyzed program violates a property using a monitor. A common property is the absence of crashes in a program. The coverage of such

a technique is typically not complete, hence it misses some program behaviors, but correct, detected bugs are true ones.

- *Model checking* [Cla97] consists in building a model of the studied program, and proving that properties hold for that model. As the model or paths can be infinite, *bounded model checking (BMC)* [CBRZ01] limits the depth of studied traces to a bound k. Bounded model checking is typically correct but not complete.
- *Symbolic execution (SE)* [CS13] is a technique that interprets a program with symbolic (non-deterministic) inputs and explores reachability properties but can be non-terminating. As paths can be infinite in practice, symbolic execution is often bounded to finite paths of length lesser than a bound k. Symbolic execution in practice is typically correct but not complete. Another approximation angle consists in concretizing some part of the exploration, system calls in particular. This variant is called *dynamic symbolic execution (DSE)* or *concolic execution* [Sen07].

The concept of completeness can be restricted to *bounded completeness*, also called *k-completeness*. An analysis is said to be k-complete when it explores all program behaviors constituting paths of length lesser than a bound k.

Note that not all program analysis techniques are either under-approximation or over-approximations. They can be neither, like machine learning techniques for software vulnerability detection [LWH+20] for instance.

3.1.3 Reachability Property

We present in this section a formalization of the reachability property, that can be assessed by a program analysis technique. We will extend this definition of reachability to include the impact of an attacker in the work presented in this thesis.

Transition System. Considering a program $\mathbb{P}$, we denote S the set of all possible states. A state is composed of the code memory, the data memory (i.e. the stack and heap), the state of registers and the location of the next instruction to execute. The set of input states of a program $\mathbb{P}$ is noted $S_0 \subseteq S$. The set of transitions (or instructions) of the program is denoted T. The execution of an instruction $t \in T$ is represented by a one-step transition relation $\rightarrow_t \in S \times S$. We denote $s \rightarrow s'$ when $s \rightarrow_t s'$ for some transition $t \in T$.

Execution Path. We extend the transition relation over any finite path $\pi \in T^*$ by the composition of a finite sequence of transitions, denoted $\rightarrow_\pi$.

Reachability. The transitive reflexive closure of $\rightarrow$ is noted $\rightarrow^*$. A state $s' \in S$ reachable in a program $\mathbb{P}$ from a state $s \in S$, is noted $s \rightarrow^* s'$. We write for two states $s, s' \in S$:

$$s \rightarrow^* s' \Leftrightarrow \exists \pi \in T^*.s \rightarrow_\pi s'$$

We use $S \rightarrow s'$ as a shortcut for $\exists s \in S.s \rightarrow s'$, and $\rightarrow_{\leq k}$ for reachability in at most k steps.

Location Reachability. We consider in the rest of the paper the case of *location reachability*: given a location l (instruction or code address) of the program under analysis, the question is whether we can reach any state s at location l. More formally, L is the finite set of locations of $\mathbb{P}$, and we consider a mapping $loc : S \mapsto L$ from states to locations. For example, loc may return the program counter value. We write $S \rightarrow^* l$ as a shortcut for $\exists s' \in S.S \rightarrow^* s' \wedge loc(s') = l$.

Definition 3.1 (Standard reachability)**.** *A location l is reachable in a program $\mathbb{P}$ if $S_0 \rightarrow^* l$.*

We extend this definition to location reachability in at most k steps, denoted:

$$S_0 \rightarrow^*_{\leq k} l$$

We now define correctness and completeness for a program analyzer.

Definition 3.2 (Correctness, completeness)**.** *Let $\mathcal{V} : (\mathbb{P}, l) \mapsto \{1, 0\}$ be a verifier taking as input a program $\mathbb{P}$ and a target location l.*

- *$\mathcal{V}$ is correct when for all $\mathbb{P}$, l, if $\mathcal{V}(\mathbb{P}, l) = 1$ then l is reachable in $\mathbb{P}$;*
- *$\mathcal{V}$ is complete when for all $\mathbb{P}$, l, if l is reachable then $\mathcal{V}(\mathbb{P}, l) = 1$;*
- *if $\mathcal{V}$ also takes an integer bound n as input, $\mathcal{V}$ is k-complete when for all bound n and $\mathbb{P}$,l, if l is reachable in at most n steps then $\mathcal{V}(\mathbb{P}, l, n) = 1$.*

3.1.4 Link to the Thesis

Part of the work described in the thesis relates to automatic program analysis, which we leverage with the reachability property in mind.

3.2 Symbolic Execution

In this section, we delve deeper into one program analysis technique, symbolic execution, as part of the work described in this thesis is based upon it. In particular, symbolic execution can assess location reachability in a program. After an overview, we present the main symbolic execution algorithm and its inherent limitations to keep in mind.

3.2.1 Overview

Symbolic execution (SE) [Kin76, GKS05, SMA05, CGP+08, CS13, BCD+18] is an automated program analysis technique for bug finding and (bounded-)verification. Symbolic execution is a symbolic exploration technique for standard reachability. SE consists in simulating the execution of a program on symbolic inputs, i.e. of undetermined value, instead of concrete ones. The analysis keeps the program state as a symbolic state where program variables are expressed as terms of symbolic inputs. The analysis records a path explored as the conjunction of constraints associated with each conditional and indirect jump taken to follow the path, called a path predicate.

SE has two main usages.

- It can assess the reachability of a code location in a path by transforming the path predicate in an SMT formula discharged to an SMT solver [BT18] that we use as a black box.
  - Either the solver answers 'SAT', meaning the formula is satisfiable and provides a model for symbolic inputs such that executing the program with those inputs will reach the code location;
  - Either the solver answers 'UNSAT' and the code location is not reachable from that path.

  - It is also possible to set a timeout to avoid having to wait a long time for the solver to answer in some edge cases. If the solver reaches the timeout then we cannot say anything about the reachability of the location.

  When used with an oracle detecting a bug, SE becomes a useful bug-finding technique, for instance, to detect use-after-free bugs [FMB+16].
- It can generate test cases with the goal of maximizing the test suite coverage, each input covering one path.

3.2.2 Symbolic Execution Algorithm

Algorithm 3.1 gives a high-level view of a typical SE algorithm, adapted for location reachability. More complex properties can be verified with the same principles, such as local predicate reachability, trace properties or hyper-properties [DBR20].

Algorithm 3.1: Standard symbolic execution algorithm, taken from [GFB21]

Input: a program P, a bound k, a target location l
Output: Boolean value indicating whether l can be reached within k steps.

```
1 for path π in GetPaths(k) do
2 |   if π reaches l then
3 |   |   Φ := GetPredicate(π)
4 |   |   if Φ is satisfiable then
5 |   |   |   return true
6 |   |   end
7 |   end
8 end
9 return false
```

The analysis follows each possible path π of a program, enumerated by the function `GetPaths`, up to a depth bound k. If π reaches the target location l, then we check whether π is indeed feasible by computing its *path predicate* Φ – a logical formula representing the path constraints over the input variables along π, and sending it to an SMT solver, that will try to answer whether the formula is satisfiable or not, and provide a model for free variables (e.g. inputs) if it is (omitted here for simplicity).

Property 3.1. *SE is correct for location reachability, and k-complete if we assume a correct encoding of path predicates [God11].*

Algorithm 3.2: Assignment evaluation in SE

Input: path predicate Φ, assignment instruction $x := expr$
Output: Updated Φ

```
1 Function eval_assign(Φ, x, expr) is
2 |   return Φ ∧ (x ≜ expr)
3 end
```

In this thesis, we will focus on the evaluation of assignments and conditional jumps for SE, detailed in Algorithms 3.2 and 3.3 respectively, as this is where our adversarial symbolic execution will mainly differ from the standard one. It requires going slightly

Algorithm 3.3: Conditional jump evaluation in SE

Input: path predicate Φ, conditional jump instruction $if\ cdt\ then\ addr_t\ else\ \ addr_e$

Data: a worklist WL containing the pending path prefixes to explore – list of pairs (path predicate, next location)

Output: WL updated in place

```
1 Function eval_conditional_jump(Φ, cdt, addr_t, addr_e) is
2     if Φ ∧ cdt is satisfiable then
3         Add (Φ ∧ cdt, addr_t) to WL
4     end
5     if Φ ∧ (¬cdt) is satisfiable then
6         Add (Φ ∧ ¬cdt, addr_e) to WL
7     end
8 end
```

deeper into details. In practice, the program paths are explored incrementally. A worklist WL records all pending paths together with their associated path predicate and their next instruction to explore.

- Assignments (Algorithm 3.2) are dealt with straightforwardly, simply adding a new logical variable definition $expr$ of the variable x to the path predicate Φ, written $x \triangleq expr$. Actually, a symbolic state usually comprises the path predicate itself plus a mapping from program variable names to logical variable names, and assignments involve both creating new logical names and updating the mapping. We abstract away from these details.
- On conditional branches (Algorithm 3.3), the symbolic path is split in two, one for each branch (cdt and $\neg cdt$), updating the path constraint Φ accordingly. The satisfiability of each associated formula is checked before adding the updated path predicates to the worklist WL.

3.2.3 Limitations

Symbolic execution suffers two main limitations hindering performance and scalability: path explosion and constraint solving.

3.2.3.1 Path Explosion

SE explores all paths, which are possibly infinitely many, even those which will prove unfeasible, hence inducing a path explosion. A program typically contains various patterns generating new paths.

- At each conditional jump, the path is split in two, one will follow the then branch, the other the else branch.
- At indirect jump instructions, possible jump targets are enumerated and a new path is created for each one.
- A loop whose termination is conditioned by a symbolic input becomes unbounded. SE unrolls the loop for each possible number of iterations, yielding possibly infinitely many new paths.
- Similarly, if a program contains a recursion whose termination is conditioned by a symbolic input, the analysis will call it for each possible recursion depth, yielding

possibly infinitely many new paths.

To tame path explosion during the exploration, different options are available:

- by identifying redundant paths that present a behavior already explored. Those can be pruned [BCE08];
- by merging paths [KP05] at the price of more complex formulas [KKBC12];
- by limiting the maximal depth of a path, hence performing bounded verification;
- by using search heuristics to guide the analysis toward exploring paths to maximize some coverage criteria [XTDHS09]. Only a subset of paths are then explored.

The last two techniques presented forgo completeness. By using such techniques, the analysis performs an under-approximation of the real program behavior and cannot be used to prove a property, but only for bug finding.

3.2.3.2 Constraint Solving

Symbolic execution generates formulas expressing the feasibility of a branch or the reachability of a location and discharges them to an SMT solver. SE analysis depends on the ability of the solver to answer those queries which tend to grow in complexity as the path explored is long and complex itself. SE performance is limited by the solver's performance, which is the main bottleneck of symbolic execution.

Analyses based on symbolic execution rely on different SMT theories, i.e. representations of structures more complex than boolean variables, such as integers or arrays, and their corresponding axioms. Some theories are notoriously hard to reason about, when they include quantifiers for instance. The work in this thesis extends a symbolic execution algorithm but does not need a more complex theory.

3.2.4 Link to the Thesis

Symbolic execution explores all program paths of bounded depth as a bug-finding technique for reachability. In this thesis, we adapt the symbolic execution engine to support fault injection and use it to assess the reachability of a code location in the presence of an advanced attacker.

3.3 Binary Code Analysis

In this section, we explore the specificities related to program analysis when applied to binary programs. We call binary code or binary program an executable composed of 0s and 1s, that the CPU will take as input. The assembly of a program, or its representation at ISA level, is binary code interpreted as instructions, in a human-readable form. The most well-known are Intel x86, ARM and RISC-V.

3.3.1 Binary VS Source Code Analysis

Binary code is a representation of what is executed on a machine, contrary to source code. Source code is a human-readable representation that contains details such as type information and data structures and hides others like memory VS register usage and flag updates. Moreover, to go from source code to binary code, compilers apply many transformation passes that:

- can remove part of the code considered to be dead code;
- can rewrite control and data flows, and rearrange the order of instructions;

- can break properties that hold at source level like constant time or non-interference;
- implements a specific behavior for each source-level undefined behavior;
- can use optimizations not guaranteed to preserve the semantics of the source code. For instance, the optimization flag '-funsafe-math-optimizations'[1] for floating point arithmetic in gcc can violate language standards.

This phenomenon is known as *What You See* (in source code) *Is Not What You eXecute* (machine code), or WYSINWYX for short [BR10]. Hence, an analysis at the binary level is closer to what is actually executed than an analysis at the source level.

Another advantage of analyzing binary code is that it is available for any program a user runs (except for remote reverse engineering). The associated source code is not always provided, in particular when analyzing commercial, off-the-shelf executables, or when studying malwares.

3.3.2 Challenges of Binary Analysis

Several challenges arise when analyzing programs at the binary level. A binary program has more lines of code than the associated source code, challenging the scaling capabilities of the analysis. Binary code is the encoding into "0"s and "1"s of an assembly code (ISA), that encompasses implicit flag update. Those updates need to be made explicit since they play an important role, for instance, in conditional jumps where they determine the choice of the jump target. While source code analysis reasons about typed variables, a binary code analysis requires precise reasoning about the memory as assignments become load and store operations. It also needs to handle explicit stack machinery. Finally, binary analysis tools tend to be harder to use than source-level code, often requiring reverse engineering skills to configure the tool and then understand the analysis results.

3.3.3 Link to the Thesis

In this thesis, we implemented our analysis technique at the binary level in order to be close to what is actually executed by the processor and how it is disturbed by fault injection.

3.4 Program Analysis Techniques for Fault Injection

In Chapter 2, we saw an advanced attacker could inject various kinds of faults into a running program. Dedicated program analysis techniques have been designed to account for those extra program behaviors in the assessment of a property. They are called SoftWare-Implemented Fault Injection techniques (SWIFI). In this section, we overview the main SWIFI techniques and their limitations.

3.4.1 Simulation

Dynamic analysis, also called *simulation* [DPdC+15, HSP20], consists in choosing concrete input and successively decoding and simulating the effects of decoded instructions along the execution path. This technique can be used to simulate the effects of fault injection. Typically, a golden run without fault is performed to identify all possible

[1] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

injection locations. Then a simulation is launched for each concrete fault-induced instruction modification. The final states are recorded and checked against the attacker's goal.

Limitation. Simulation only supports concrete inputs, forcing the analysis to only consider one reference execution path at the time, hence is not complete. Furthermore, simulation is limited by the explosion of simulations to run in particular when considering an attacker able to perform multiple faults, faults with different effects, or as the program grows in size.

3.4.2 Mutant Generation

The *mutant generation* approach [CCG13, RG14, GWJLL17, CDFG18, GWJL20] consists in analyzing slightly modified versions of the program (named mutants), each of them embedding a different faulty instruction. For example, Figure 3.3 illustrates how a program containing few instructions (Figure 3.3a) can be faulted into three different mutants 3.3b, 3.3c and 3.3d. Each mutant is then analyzed on its own using standard

```
a := function1()
b := x + y * z
c := 42
```

(a) Original

```
a := fault_value
b := x + y * z
c := 42
```

(b) Mutant 1

```
a := function1()
b := fault_value
c := 42
```

(c) Mutant 2

```
a := function1()
b := x + y * z
c := fault_value
```

(d) Mutant 3

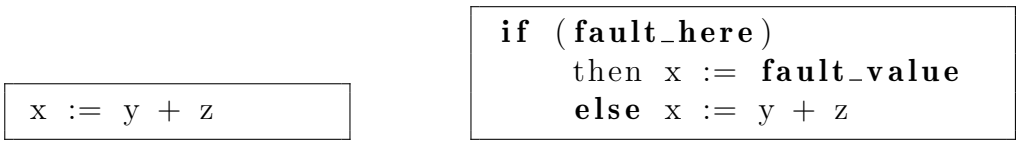
Figure 3.3: Mutant generation transformation in pseudo-code

program analysis tools. Inputs can be symbolic or concrete. Mutant generation is usually used only for single-fault analysis.

Limitation. The main limitation of mutant generation is the explosion of mutants, in particular when considering an attacker able to perform multiple faults, faults with different effects, or as the program grows in size. Also, as the different mutants differ only slightly, analyzing each of them separately wastes lots of time repeating similar reasoning.

3.4.3 Forking Techniques

The *forking approach* [PMPD14, BBC+14, BHE+19, LFBP21, Lan22] consists in instrumenting the analysis (or the code, via instrumentation) to add all possible faults as forking points (branches) controlled by boolean values indicating whether a particular fault will be taken or not, plus constraints on the maximal number of faults allowed. A forking data fault is illustrated in Figure 3.4. The original statement (Figure 3.4a) is transformed into an if-then-else-like construct (Figure 3.4b). A standard program analysis technique is then launched – typically symbolic execution or bounded model checking. Compared with the mutant generation technique, this method allows sharing of parts of the analysis between the different possible faults.



(a) Original statement

(b) Forking transformation

Figure 3.4: Forking technique transformation in pseudo-code

Limitation. The number of paths explored by the analysis explodes with the number of possible faults (forking points).

3.4.4 Related Work

We list in the following the main topics related to software-implemented fault injection, namely robustness analysis and mutation testing.

3.4.4.1 Robustness Analysis

SWIFI is also used for robustness evaluation [LH07, PNKI08, HTS+17, WTSS13, LHGD18, PLFP19, CDSLN20, ZGWL+21], to consider the impact of *safety faults* instead of security related faults. This type of analysis aims to verify the correct behavior of error-handling mechanisms, implemented to recover from, or be resilient to, such safety faults. They can be generated by hardware damage, nuclear radiation or cosmic rays for instance. Robustness analyses rely also on mutant generation or forking techniques. The fault models are similar to hardware fault injection, yet multi-fault is not really an issue there, as faults are supposed to originate from safety issues and have no reason to accumulate unreasonably.

3.4.4.2 Mutation Testing

Sometimes called software fault injection, mutation testing [PIK+18, CN13] aims to generate a comprehensive test suite by building test cases discriminating various mutants of a program, and is recognized as a very powerful testing criterion. As it focuses on coverage, the mutant explosion cannot be avoided. Dedicated SE techniques [PM10, BCDK14, BKC14, MBK+18] have been designed.

3.4.5 Link to the Thesis

In this thesis, we present a new program analysis technique to represent the effect of fault injection on a program, with the aim of mitigating the state explosion faced by the existing techniques.

Chapter 4

Adversarial Reachability

Contents

In this chapter, we propose our model of an advanced attacker in Section 4.1, encompassing capabilities listed in Chapter 2. We describe a new formalization to represent the impact of this advanced attacker on a program, called *adversarial reachability*, in Section 4.2. Then, we design an algorithm to evaluate adversarial reachability, *forkless adversarial symbolic execution* (FASE) in Section 4.3. The challenge is to prevent the path explosion faced by the state of the art. We answer this challenge with a dedicated *forkless encoding* of faults (Section 4.3.2) and optimizations (Section 4.4) aiming to reduce the number of injections without loss of generality. Discussions and comparisons with related work are presented in Sections 4.5 and 4.6. The practical implementation of the algorithms described in this chapter is detailed in Chapter 5.

Part of the work presented in this chapter has been published at ESOP 2023 [DBP23].

4.1 Attacker Model

This section presents how we model an advanced attacker at program level. Advanced attackers can do more than carefully craft legitimate inputs to trigger vulnerabilities in software. They can use a wide variety of attack vectors (e.g. hardware fault injection attacks, software-implemented hardware attacks, micro-architectural attacks, software attacks, etc), in any combination, and multiple times. We suppose attack vectors prerequisites have been met, and only consider the impact of the generated faults on the program under attack.

4.1.1 Advanced Attacker

We start by providing a definition of what we mean by an advanced attacker at the program level.

Definition 4.1. *We define an advanced attacker as an attacker able to perform multiple program-level actions during one attack, independently of the actual attack vector used.*

We model an advanced attacker by an attacker model composed of the following components:

- a set of attacker actions;
- a fault budget;
- an attacker goal.

We describe those components in more detail in the following sections.

4.1.2 Attacker Actions

The first element of our attacker model is a set of attacker actions. Its purpose is to establish clear boundaries of the power of the attacker model on a program so it can fit a wide variety of security scenarios. An attacker action is composed of the following elements:

- It encompasses the *type of modification* brought by the attack action, such as flipping a bit, changing a word or resetting a variable;
- An action is also defined by the *object* it is applied to. It can be a register, an instruction op-code or a cell in memory for instance.

Actions can also be described with a negative definition, that is to say, they can be explicitly restricted. For instance, if a strong memory isolation is supposed to be in place, it can be interesting to restrict a memory write capability to a certain range of addresses.

We suppose an attacker can leverage multiple distinct attacker actions, possibly originating from different attack vectors, hence the attacker model includes a *set* of attacker actions.

Attacker actions are similar to fault models used to describe the effects of a hardware fault injection on a program. Hence, in the rest of this manuscript, we will use interchangeably *attacker capability*, *attacker action* and *fault model*.

4.1.3 Fault Budget

Now that our attacker has different actions at their disposal, they can perform them at will on the program. To restrict the power of the attacker model we introduce a

maximum number of actions that can be performed during one attack. We call it the *fault budget* of the attacker model. An attacker with infinite power can do anything with the program, including disarming potential protections, which is not always very interesting to consider or realistic in a threat assessment.

4.1.4 Attacker Goal

Finally, the attacker is trying to achieve something with their attack. We model this attack goal by an oracle over the state of the program, and the aim of the attacker is to reach a state of the program satisfying the oracle. A simple oracle is a code location the attacker may want to reach. More complex functions of the program state can be defined such as properties of the program's variables or trace properties.

The ability to reach a state satisfying a property is the standard notion of *reachability*. However, here, the attacker can disrupt the execution of the program with their capabilities in order to reach such a state. Hence we define the ability to reach a program state satisfying a property in the presence of an advanced attacker as *adversarial reachability*. This concept is detailed in Section 4.2.

4.2 Adversarial Reachability

In this section, we describe our formalism to reason about a program's execution in the presence of an advanced attacker. We consider a program as a transition system, where the attacker's capabilities are expressed as new transitions. Adversarial reachability is standard reachability in this modified transition system. We start this section with a reminder of the notations used in Section 3.1.3 for standard reachability, then we define adversarial reachability and related properties.

4.2.1 Reminder: Standard Reachability

We use the following notations to describe a program as a transition system and express standard reachability:

- a program $\mathbb{P}$;
- the set S of all possible states, S_0 the set of initial states of $\mathbb{P}$;
- the set T of transitions (or instructions) of the program $\mathbb{P}$;
- the execution of an instruction $s \to s'$ for some transition $t \in T$, with $s, s' \in S$;
- an execution path $\to_\pi$, with $\pi \in T^*$;
- the standard reachability of a state $s \in S$ is $S_0 \to^* s$, as a short way to write $\exists s_0 \in S_0 . s_0 \to^* s$;
- the standard reachability of a location $l \in L$ is $S_0 \to^* l$ as a short way of writing $\exists s \in S . S_0 \to^* s \wedge loc(s) = l$.

We refer the reader to Section 3.1.3 for the definition of correctness and completeness for standard reachability.

4.2.2 Adversarial Reachability Definition

We now build upon the transition system and notations previously described to include the presence of an advanced attacker and detail how attacker actions are encoded as faults.

4.2.2.1 Advarsarial Transitions

Adversarial Transitions. We extend the transition model previously described to include *adversarial transitions* denoted $\rightsquigarrow_{\mathcal{A}} \in S \times S$, related to an attacker model $\mathcal{A}$. The total set of possible transitions in the context of an attacked program is composed of legitimate program transitions and attacker transitions, noted $T_{\mathcal{A}} = T \cup \rightsquigarrow_{\mathcal{A}}$.

Then, the transition relation of $\mathbb{P}$ under attack from an attacker model $\mathcal{A}$ is denoted as $\mapsto_{\mathcal{A}} = \rightarrow \cup \rightsquigarrow_{\mathcal{A}} = (\cup_{t \in T} t) \cup \rightsquigarrow_{\mathcal{A}}$.

Remark. The successor of a state in the execution of the program has no longer a unique solution, as either the standard transition $\rightarrow$ corresponding to the legitimate program instruction, or one or more adversarial transitions $\rightsquigarrow_{\mathcal{A}}$ can be taken.

Adversarial Path. We denote $\pi \in T_{\mathcal{A}}^{*}$, an *adversarial path*, representing a succession of program instructions and adversarial actions.

Program States. We consider a program state $s \in S$ to be composed of:

- the code memory M_c,
- the data memory (i.e. the stack and heap) M_d,
- the set of registers R,
- the location of the next instruction to execute ip.

The location of the next instruction to execute is often contained in a register, we single it out in this model for clarity.

Transitions Granularity. A transition, adversarial or not, can only modify the next instruction pointer and one other element of the state. For instance, we do not consider writing at multiple locations in the data memory as one transition.

4.2.2.2 Fault Expression

We now illustrate various types of faults and how they can be represented as adversarial transitions.

Faults In-Between Instructions. To represent a fault happening between normal transitions, in addition to legitimate program behavior, the adversarial transition corrupts the relevant part of the program state, but leaves the next instruction pointer unchanged.

We take the example of data faults corrupting a value in the data memory, with $addr$ an address in the data memory and $fault\_v$ a corrupted value.

$$s\{M_c, M_d, R, ip\} \rightsquigarrow s'\{M_c, M_d\{addr \leftarrow fault\_v\}, R, ip\}$$

This fault model corresponds for instance to a Rowhammer bit-flip, happening in memory independently of the program execution.

Depending on the grammar complexity of the language modeled by the transition system, faults in-between can have varying expressivity.

- For instance, in assembly, where every load and intermediate operation result are stored in intermediary variables, typically in registers, a fault in-between assembly instructions can express all possible combinations of corruption of the end result;
- On the opposite, if complex operation aggregations are allowed by the grammar, having a load operation as one element of a binary operation without intermediate storing for instance, then an in-between fault could not fault just the result of

the load, hence cannot express corruptions between the memory and the data treatment unit.

Specific Fault Behavior. To represent a specific behavior more precisely, the corrupted value can be expressed as a function. Here, we denote $Fault$ such a function.

$$s\{M_c, M_d, R, ip\} \rightsquigarrow s'\{M_c, M_d\{addr \leftarrow \text{Fault}(M_d[addr])\}, R, ip\}$$

$$s_{i-1} \rightarrow_{t_i} s_i \rightarrow_{t_{i+1}} s_{i+1} \rightarrow_{t_{i+2}} s_{i+2}$$

$$s_{i-1} \rightarrow_{t_i} s_i \rightsquigarrow s'_i \rightarrow_{t_{i+1}} s'_{i+1} \rightarrow_{t_{i+2}} s'_{i+2}$$

$$s_{i-1} \rightarrow_{t_i} s_i \rightsquigarrow s'_{i+1} \rightarrow_{t_{i+2}} s'_{i+2}$$

To precisely express the bit-flip from Rowhammer, we take

$$\text{Fault}(v) = v \oplus (1 << bv)$$

with v a value, $\oplus$ denoting a bitwise xor operator and bv indicating the bit that is flipped.

We now discuss the impact of using a $Fault$ function expressing the fault behavior on in-between fault expressivity:

- If the $Fault$ function cannot access the inner representation of the expression behind v, typically explore its syntax tree, but only use v as such, then expressivity restrictions discussed above for in-between faults remain valid;
- On the contrary, if the $Fault$ function can access and modify the inner representation of v, restrictions are lifted. Coming back to the load inside a binary operation example, such a $Fault$ function could fault only the load part, equivalent to faulting a solitary intermediate load.

Faults Replacing an Instruction. On the opposite of faults in-between instructions, faults can be modeled as replacing a legitimate instruction, changing its original effect. The adversarial transition 'hijacks' a legitimate transition, mimicking it but with faulty behavior, and updating the instruction pointer to directly go to the next instruction, in effect bypassing the legitimate transition.

Staying with the data fault in memory example, a legitimate store instruction of the value contained in the register r, at the address $addr$, written

$$s\{M_c, M_d, R, ip\} \rightarrow s'\{M_c, M_d\{addr \leftarrow R[r]\}, R, ip \leftarrow ip + 1\}$$

can be replaced by the following adversarial transition

$$s\{M_c, M_d, R, ip\} \rightsquigarrow s'\{M_c, M_d\{addr \leftarrow fault\_value\}, R, ip \leftarrow ip + 1\}$$

This type of fault represents for instance the corruption of a stored value before it reaches the memory.

Most in-between faults can be equivalently expressed as a replacing fault. The same expressivity considerations apply.

- As in-between faults are somewhat independent of the execution, as long as they happen before the corrupted value usage, they can be pushed ahead to the previous store. A replacing fault can then be used instead;

- Only data in the initial state cannot be corrupted with a replacing fault, as no instruction is storing them. However, each usage of this initial data can be faulted with a replacing fault;
- Delaying the fault until the corrupted value usage and using an adversarial transition of each usage of that value until it is overwritten, if at all (global initial variables are often constants), is equivalent to in-between faults. However, this requires much more faults in the attacker's budget.

Transient & Permanent Faults. Transient faults refer to faults whose effects last only until the affected value is overwritten. On the contrary, permanent faults are faults that cannot be overwritten during the execution of the program. This can represent permanent damage in the chip containing code or data memory, or faults in the code memory when it is not changed during the execution of the program, which is usually the case, except for self-modifying programs.

The formalization presented is well-suited for transient faults, as illustrated by in-between and replacing faults previously described. We can also express permanent faults in the code memory, assuming it is unchanged during the execution.

$$s\{M_c, M_d, R, ip\} \rightsquigarrow s'\{M_c\{addr \leftarrow fault\_instr\}, M_d, R, ip\}$$

The new instruction $fault\_instr$ will be used instead of the original one for the remainder of the execution.

However, the state model used does not include the memory physical layer and thus prevents us from expressing permanent fault on other parts of the program state like the data memory or registers. Hence, this formalism cannot express faults such as stuck-at bits (except having an in-between fault applying the fault effect each time). A possibility to include permanent faults on data would be to add a mapping of the physical layer to the values in the state, where a mask and fault effects would be applied to each read and write of values.

4.2.2.3 State-of-the-art Fault Models

To specify practical fault models, restrictions are applied onto $\rightsquigarrow_{\mathcal{A}}$, limiting what part of the state can be modified and how. In particular, relevant $Fault$ functions specifying the fault behavior can be defined for each fault model available to the attacker. We now explore a few common fault models.

Data Faults. Data faults have been described above, for arbitrary data faults and bit-flips. Extensions to other types of data faults can be derived from those. We write here a reset fault on the register r.

$$s\{M_c, M_d, R, ip\} \rightsquigarrow s'\{M_c, M_d, R\{r \leftarrow 0\}, ip \leftarrow ip + 1\}$$

Instruction Skip Fault. An instruction skip consists of bypassing any sequence of consecutive instructions with the following adversarial transition.

$$s\{M_c, M_d, R, ip\} \rightsquigarrow s'\{M_c, M_d, R, ip \leftarrow ip + n\}$$

We write here $ip+n$ to denote the nth next instruction address in the code memory layout. A single instruction skip is written for $n = 1$.

Interestingly, this representation of instruction skips can model the corruption of the program counter, modeled by ip here. But it also encompasses replacing an instruction

with a no-operation (NOP), as the proposed adversarial transition does not modify the state and goes to the next instruction.

Replaying Instructions. Some observations of instructions skips report a replay of the previous instructions, in particular for attacks on the instructions prefetch buffer. For instance, to skip 4 instructions n to $n+3$, the instructions up to $n-1$ are executed normally, and then instructions $n-4$ to $n-1$ are executed again, before going straight to instruction $n+4$. This can be represented with 2 adversarial transitions, one setting $ip \leftarrow ip-4$ just before the instruction n should be executed, and one adversarial transition $ip \leftarrow ip+4$ just before the instruction n should again be played.

However, for those two adversarial transitions to be considered as one fault, extra elements need to be added to the program state to track the relation between instructions.

Test Inversion. We consider a conditional jump to be written as follows, with a conditional expression written $(a\ ?\ b : c)$ where if a is true, then the expression evaluates to b and to c otherwise. ip_A and ip_B are two instructions pointers.

$$s\{M_c, M_d, R, ip\} \rightarrow s'\{M_c, M_d, R, ip \leftarrow R[r]\ ?\ ip_A\ :\ ip_B\}$$

The fault model consisting of inverting a test can be written multiple ways, corrupting the test condition before the conditional jump for instance.

$$s\{M_c, M_d, R, ip\} \rightsquigarrow s'\{M_c, M_d, R\{r \leftarrow \neg r\}, ip\}$$

Or the conditional jump can be replaced with an adversarial conditional jump inverting the instructions pointers.

$$s\{M_c, M_d, R, ip\} \rightsquigarrow s'\{M_c, M_d, R, ip \leftarrow R[r]\ ?\ ip_B\ :\ ip_A\}$$

The test inversion fault model illustrates that one fault can be expressed with different adversarial transitions.

Limits and Extensions. We discuss now some limits of the current formalization and possible extensions.

- Complex, multi-effect fault models can be split into their atomic components, modeled separately. However, the number of fault count needs to account for this split. Moreover, synchronizations between those atomic adversarial transitions may require extending the state with some new elements, counters for instance;
- This formalization does not include any micro-architectural detail. As discussed for instruction replay and permanent data faults, the representation of the program state can be extended to account for more fault models.

4.2.2.4 Attacker Budget

Still, we need to take into account the maximum number of faults the attacker can perform along an execution path. Given a path π over $T^*_{\mathcal{A}}$, π is said to be *legit* if it does not contain $\rightsquigarrow_{\mathcal{A}}$, and *faulty* otherwise. The number of occurrences of transitions $\rightsquigarrow_{\mathcal{A}}$ in π is its *number of faults*.

Given a bound $m_{\mathcal{A}}$ on the fault capability of $\mathcal{A}$, we define $\mapsto^*_{(\mathcal{A},m_{\mathcal{A}})}$ by limiting the adversarial reachability relation to paths π with less than $m_{\mathcal{A}}$ faults. We consider $m_{\mathcal{A}}$ to be $+\infty$ in case the attacker has no such limitation. For the sake of simplicity, in the following, we will consider $m_{\mathcal{A}}$ as an implicit parameter of $\mathcal{A}$, and simply write $\mapsto^*_{\mathcal{A}}$ instead of $\mapsto^*_{(\mathcal{A},m_{\mathcal{A}})}$.

4.2.2.5 Adversarial Reachability

We extend the notations for state reachability to the relation $\mapsto_{\mathcal{A}}$. Especially, $S \mapsto^*_{\mathcal{A}} s'$ means $\exists s \in S.s \mapsto^*_{\mathcal{A}} s'$. The adversarial transition relation up to k is denoted $\mapsto_{\mathcal{A},\leq k}$ and reaching a location l from a set of states S will be denoted $S_0 \mapsto^*_{\mathcal{A}} l$.

Definition 4.2 (Adversarial reachability)**.** *Given an attacker $\mathcal{A}$ with a $m_{\mathcal{A}}$ fault budget and a program $\mathbb{P}$, a location $l \in L$ is adversarially reachable if $S_0 \mapsto^*_{\mathcal{A}} s' \wedge loc(s') = l$ for some $s' \in S$.*

In the following, the adversarial reachability of location l from a set of states S will be denoted $S_0 \mapsto^*_{\mathcal{A}} l$.

We extend the adversarial reachability definition to adversarial reachability in at most k steps, noted $S_0 \mapsto^*_{\mathcal{A},\leq k} l$.

4.2.2.6 Attacker Model Formalization

Finally, we can write the formalization of our attacker model representing an advanced attacker using adversarial reachability.

Definition 4.3 (Attacker model)**.** *We define an attacker model in this formalism by*

$$\mathcal{A} = \{\{\rightsquigarrow\}, m_{\mathcal{A}}, \phi\}$$

- *the adversarial transitions $\{\rightsquigarrow\}$ correspond to attacker capabilities;*
- *$m_{\mathcal{A}}$ is the maximum number of faults allowed;*
- *the goal of the attacker is noted $\phi \in S \times \{0,1\}$. If $\exists s \in S$ such that s is adversarially reachable and $\phi(s) = 1$, then the attack is successful. For location reachability of a location $l_{\mathcal{A}}$, we take $\phi(s) = (loc(s) == l_{\mathcal{A}})$.*

4.2.3 Properties

We now discuss the properties of adversarial reachability and of a verifier for adversarial reachability.

Proposition 4.1. *Standard reachability implies adversarial reachability. The converse does not hold.*

Proof. Standard reachability can be viewed as adversarial reachability with an attacker able to perform 0 faults. □

We redefine what it means for an analysis answering adversarial reachability to be correct, complete and k-complete.

Definition 4.4. *Let $\mathcal{V}_A : (\mathbb{P}, A, l) \mapsto \{1, 0\}$ be a verifier taking as input a program $\mathbb{P}$, an attacker A with m_A fault budget and a target location l.*

- *$\mathcal{V}_A$ is correct given A when for all $\mathbb{P}$, l, if $\mathcal{V}_A(\mathbb{P}, A, l) = 1$ then l is adversarially reachable in $\mathbb{P}$ for attacker A;*
- *$\mathcal{V}_A$ is complete given A when for all $\mathbb{P}$, l, if l is adversarially reachable for attacker A then $\mathcal{V}_A(\mathbb{P}, A, l) = 1$;*
- *if $\mathcal{V}_A$ also takes an integer bound n as input, $\mathcal{V}_A$ is k-complete given A when for all integer n and $\mathbb{P}$,l, if l is adversarially reachable in at most n steps then $\mathcal{V}_A(\mathbb{P}, A, l, n) = 1$.*

4.2.4 Discussion

In this section, we take a step back and discuss what kind of properties can be verified using our formalization and how it could be extended to other types of attacker actions. We also discuss the advantages of studying location reachability.

4.2.4.1 Other Properties

Our formalism itself is quite generic and can accommodate a wide range of properties, as we mainly keep the property unchanged but modify the underlying transition system. We focus in this work on the reachability property in this extended transition system. However, more complex properties than location reachability can be studied. We list here some examples.

- We can have an oracle over the state more complex, for instance looking for use-after-free or buffer overflows.
- The oracle could be a monitor for runtime errors.
- We could imagine an attacker willing to activate a non-terminating execution (denial of service).

Any property studied based on a transition system could be studied in the presence of an attacker with our extended transition system.

4.2.4.2 Other Attacker Actions

The proposed formalism only considers active faults, i.e. faults modifying the state. It does not include the possibility of passive faults, i.e. an attacker read action, nor an attacker able to execute multiple times a program and learn from it until they can achieve their goal.

While those are out of the scope of this thesis, we believe they could benefit from our formalization, by extending the attacker model with a knowledge component, and the transition system with a leakage model. Those extensions would allow to represent an attacker performing combined passive and active attacks [AVFM07].

4.2.4.3 Location Reachability

We want to stress that while location reachability can be seen as a basic case, we consider it sufficient here for two reasons:

- first, it keeps the formalism light while still straightforward to generalize to stronger reachability properties (e.g., local predicates of the form (l, φ), sets of finite traces, etc.);
- second, it is already rather powerful on its own, as we can still instrument the code to reduce some stronger forms of reachability to it (e.g., adding local assertions or monitors).

4.2.5 Conclusion

We proposed a formalization of an advanced attacker and their impact on a program. We detail the concept of adversarial reachability as an attacker's goal. Modeling advanced attackers allows to reason about them and ultimately strengthen programs against those attackers. We also defined what it means for a verifier to be correct and complete with respect to adversarial reachability. In the following section, we propose a technique to assess adversarial reachability in practice.

4.3 Forkless Adversarial Symbolic Execution (FASE)

In this section, we present our algorithm to verify adversarial reachability. It aims to represent in practice the impact of an advanced attacker on a program. We mitigate the path explosion faced by existing techniques (Section 3.4) with a forkless encoding of attacker actions.

4.3.1 Overview

We conceived our fault injection technique with the following design guidelines in mind.

- We want a technique that *prevents the path explosion* faced by the state-of-the-art, and is *generic* in supported fault models. We introduce in Section 4.3.2 the forkless fault encoding that embedded the activation and effect of a fault in a non-forking manner, and that is able to support many different fault models;
- We want an automated technique that is *correct* and *k-complete* for adversarial reachability. We achieve this by basing our technique on symbolic execution as described in Section 4.3.3, which is correct and k-complete for standard reachability;
- We want to *limit the added complexity of generated SMT queries*, as query resolution is the main bottleneck of symbolic execution, which we burden with more complexity due to the fault encodings. We designed two optimizations presented in Section 4.4, Early Detection of fault Saturation (EDS) and Injection On Demand (IOD) to reduce the number of fault injections in SMT formulas.

4.3.2 Forkless Fault Encoding

We focus now on our forkless encoding of faults and the fault models it supports. It is using this encoding that we will integrate the attacker's actions into our analysis.

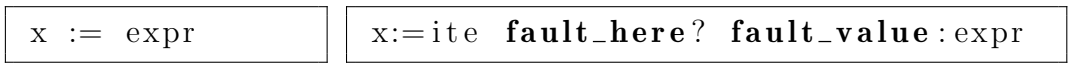
4.3.2.1 Overview

We use the term fault encoding to denote the replacement of a program instruction (or group of) by some other instruction(s) with a different, illegitimate, behavior.

The forkless encoding aims to address the path explosion induced by the forking treatment of fault injection in prior works. It focuses on data corruption, which can be values or addresses, leading to a wide variety of supported fault models. The forkless encoding consists of wrapping arithmetically an expression and embedding the activation of the fault and its effect.

- The **fault activation** is a boolean variable whose value conditions if the fault is active, i.e. modifies the normal program behavior, or if it is inactive and can be considered as not present. A new activation variable is generated for each possible fault location;
- The **fault effect** is a value corresponding to the corruption applied to the program and depends on the fault model.

We show in Figure 4.1 an arbitrary data fault on the assignment of x receiving the expression $expr$. The encoding uses an *ite* operator, an inlined form of if-then-else. The activation of this fault location is determined by the symbolic boolean value $fault\_here$, and the corrupted value of x, i.e. the fault effect, is the fresh variable $fault\_value$.



(a) Original statement (b) Forkless transformation for arbitrary data fault

Figure 4.1: Forkless injection technique

The point is to embed the fault injection as an arithmetical expression inside the path predicate, *without any explicit path forking* at the analysis level, in order to let the analyzer reason about both legit executions and faulty executions at the same time – this is akin to path merging in some ways, except that we do it only for the treatment of fault injection (we could also see the approach as avoiding undue path splits).

4.3.2.2 Different Forkless Encodings

Different forkless encodings are possible. We considered a few different encodings for arbitrary data faults in Table 4.1, leveraging various operators:

- the inlined if-the-else operator (Ite),
- the multiplication operator $*$ (Mul),
- the bitwise and operator & (And),
- the bitwise xor operator $\oplus$ (Xor).

Table 4.1: Forkless encoding variants for arbitrary data faults

Fault model	Encoding
Original	$x := expr$
Ite	$x :=$ ite $fault\_here$? $fault\_value$: $expr$
Mul	$x := expr + fault\_here * fault\_value$
And	$x := expr +$ (-$fault\_here$) & $fault\_value$
Xor	$x := expr \oplus ((-fault\_here)$ & $fault\_value)$

We use $(-fault\_here)$ as a shortcut for a signed extension of the boolean variable $fault\_here$ and then take its opposite. When the fault is active and $fault\_here := 0b1$, we consider $(-fault\_here)$ to be 0xffffffff (only ones in the binary representation, equivalent to -1 for signed integers), serving as a mask for the fault effect. The opposite of 0 remains 0.

Experiments (see Appendix B.5) show that different solvers slightly favor different encodings in terms of performance. For the solver used (see Section 5.2.2.4), the ite operator proves to be the most efficient. Hence, we chose to mainly use the *ite* expression operator, an inlined form of if-then-else at the expression level that does not induce forks in the analysis.

4.3.2.3 Supported Fault Models

We mainly illustrated the forkless fault encoding with arbitrary data faults, but other fault models are supported, summarized as pseudo code transformation in Tables 4.2.

Link With Adversarial Transitions. In general, our forkless fault models are implementations of replacing adversarial transitions (see Section 4.2.2.2), hijacking a legitimate instruction with a faulty behavior. By using in-between faults, an algorithm would require to consider faulting each possible memory cell and register between each instruction. This is much more computationally heavy than replacing instructions with

Table 4.2: Forkless encodings for various fault models

Fault model	original instruction	Forkless encoding
Arbitrary data	$x := expr$	$x := ite\ fault\_here\ ?\ fault\_value\ :\ expr$
Variable reset	$x := expr$	$x := ite\ fault\_here\ ?\ 0x00000000\ :\ expr$
Variable set	$x := expr$	$x := ite\ fault\_here\ ?\ 0xffffffff\ :\ expr$
Bit-flip	$x := expr$	$x := ite\ fault\_here\ ?$ $(expr\ xor\ 1 << fault\_value)\ :\ expr$
Test inversion	$if\ cdt$ $then\ goto\ addr_1$ $else\ goto\ addr_2$	$if\ (ite\ fault\_here\ ?\ \neg cdt\ :\ cdt)$ $then\ goto\ addr_1$ $else\ goto\ addr_2$
Instruction skip	$x := expr$	$x := ite\ fault\_here\ ?\ x\ :\ expr$
	$jump\ addr$	$if\ fault\_here\ then\ jump\ next$ $else\ jump\ addr$

predefined behavior expressing specific fault models. We detail, for each fault model, the link with adversarial transitions (examples of fault models are detailed in Section 4.2.2.3).

Arbitrary Data Fault. This fault model only targets assignment operations, with an arbitrary effect, meaning $fault\_value$ is a fresh variable the solver will propose an interesting value for. Arbitrary data faults impact the data memory (load and store operations) and registers. Arbitrary data faults implement adversarial transitions described in Section 4.2.2.2.

Reset Fault. We can use the forkless encoding to reset a variable instead of the legitimate value update. Here, the effect is simply to propagate the value 0. Reset faults impact the data memory (load and store operations) and registers, as illustrated in Section 4.2.2.3.

Set Fault. Similarly to reset faults, the forkless encoding can instantiate a set fault by propagating the value 0xffffffff (all bits at one in the binary representation).

Bit-flip Fault. We encode a bit-flip as a xor of the original expression with a mask containing only one set bit at a place determined by $fault\_value$. Bit-flip faults impact the data memory (load and store operations) and registers, as described in Section 4.2.2.2.

Test Inversion Fault. We consider the test cdt of a conditional jump as a data and fault it with our forkless encoding. The effect of the fault is to take the negation of the condition, $\neg cdt$, hence inverting the test. We write $goto\ addr_1$ and $goto\ addr_2$ to represent the execution continuing to different instructions depending on the branch taken. Here we chose a compromise between the first adversarial transition proposed (corrupting the condition beforehand) and the second (inverting the addresses) discussed in Section 4.2.2.3. The first one could be confused with a data fault that, interestingly, can have a test inversion effect. With the expressive grammar we have in practice, we can express the negation of the condition inside the if, which is equivalent to inverting jump addresses.

Instruction Skip Fault. We can extend the forkless encoding to support the instruction skip fault model by also considering jump addresses as data. With this representation, we perform instruction skip as NOPs, and not as a corruption in the program counter.

All categories of instructions have to be faulted to simulate their skipping. We take our usual data fault encoding for assignments, with the old value of the left-hand side as the fault effect. We transform jumps into conditional jumps where, if the fault is active, the execution goes to the next address in the memory. More detail about the implementation of the instruction skip fault model across instructions are available in Section 5.3.3.3.

Other Fault Models. The list of fault models above does not represent exhaustively what can be encoded with a forkless encoding. It showcases the most well-known ones and can be extended to other fault models. We discuss encoding other forkless fault models in Section 4.5.1.

4.3.2.4 Forkless Faults in the Analysis

For each instruction that could be targeted by the attacker, a forkless fault is injected by the analysis. This allows to consider all possible placings simultaneously. Only later will it be decided which faults are interesting and can lead to a successful attack path. It also enables *a multiple fault analysis* by choosing how many faults are active at the same time in a path, without any extra path created.

4.3.2.5 Complexity Trade-off

While a forkless encoding indeed allows a significant path reduction compared to forking approaches, the corresponding path predicates are more complicated than standard path predicates, as they involve lots of extra-symbolic variables for deciding whether the faults occur and emulating their effect. This is experimentally verified in Chapter 6. We show later in Section 4.4 how to reduce these extra variables through two dedicated optimizations.

Also, note that we used the expressivity of the underlying SMT theory to encode the possibility of faults and their effects. A more powerful, and hence more complex to solve, theory is not needed here.

4.3.3 Fault Injection Algorithm

We present now how we use the forkless fault encoding to modify a symbolic execution algorithm into adversarial symbolic execution, an algorithm verifying adversarial reachability.

4.3.3.1 Overview

In the same way that we extended a transition system with adversarial transitions to go from expressing standard reachability to adversarial reachability, we extend symbolic execution, a correct and k-complete analysis technique for standard reachability, with a fault encoding to build an analysis correct and k-complete for adversarial reachability.

Our extension of the symbolic execution algorithm mainly consists of embedding our fault encoding in the path predicate instead of the normal behavior of the program. Are concerned the evaluation of assignments and conditional jumps. The various aspects of symbolic execution not discussed in the following are kept as is. We named our technique Forkless Adversarial Symbolic Execution, FASE for short, to express the use of our forkless encoding (Forkless) to prevent path explosion in representing an advanced attacker (Adversarial) in a reachability-oriented technique (SE). We also use

the name Adversarial Symbolic Execution (ASE) to designate integrating faults into a symbolic execution algorithm, without explicitly mentioning whether they are forkless or forking faults.

4.3.3.2 Building the Adversarial Path Predicate

We start by considering data faults. FASE requires modifications to the evaluation of assignments and conditional jumps.

Fault Counter. We keep track of the number of active faults with a fault counter nb_f, containing the sum of all boolean activation values, extended to integers.

Adversarial Reachability Evaluation. The main algorithm for adversarial symbolic execution (Algorithm 4.1) changes compared to standard SE (Algorithm 3.1) to include the impact of the attacker. The function `GetAdversarialPaths` (line 1) is the adversarial counterpart of `GetPaths`, enumerating all possible adversarial paths of bounded depth. The function `GetPredicate` (line 3) then computes the adversarial path predicate for each adversarial path. The adversarial reachability of a code location l also ensures the attacker has not exceeded their fault budget ($nb_f \leq m_{\mathcal{A}}$) line 4.

Algorithm 4.1: Adversarial symbolic execution algorithm

Input: a program P, a bound k, a target location l, a maximal number of faults $m_{\mathcal{A}}$
Data: fault counter nb_f
Output: Boolean value indicating whether l can be reached within k steps.

```
1 for path π in GetAdversarialPaths(k) do
2     if π reaches l then
3         Φ := GetPredicate(π)
4         if Φ ∧ (nb_f ≤ m_A) is satisfiable then
5             return true
6         end
7     end
8 end
9 return false
```

Adversarial Assignment Evaluation. The assignment evaluation, originally described in Algorithm 3.2 for standard SE, is transformed as illustrated in Algorithm 4.2. We consider an assignment of the form 'a variable x receives an expression $expr$', written $x := expr$. The assignment evaluation process embeds a wrapper, `FaultEncoding`, encoding the fault in a forkless manner. It involves the declaration of fresh symbolic variables for fault activation and fault effects – hence the update of the path predicate Φ. The fault counter nb_f is updated by adding to it an extension of the boolean activation variable $nb_f := nb_f + fault\_here$. A new expression containing a fault injection, $expr'$, is computed. The `eval_assign` function returns the updated path predicate Φ, augmented with the assignment of x to the faulted expression.

Adversarial Conditional Jump Evaluation. The conditional jump evaluation, originally described in Algorithm 3.3 for SE, is transformed as illustrated in Algorithm 4.3. We consider a conditional jump of the form 'if a condition cdt evaluates to true, then the branch continuing at address $addr_t$ is taken, otherwise, the branch at $addr_e$ is

Algorithm 4.2: Forkless assignment evaluation for data faults

Input: path predicate Φ, assignment instruction $x := expr$
Data: fault counter nb_f
Output: updated Φ, nb_f updated in place

```
1 Function eval_assign(Φ, x, expr) is
2 |   Φ', expr', nb_f := FaultEncoding(Φ, expr, nb_f)
3 |   return Φ' ∧ (x ≜ expr')
4 end
```

taken', written $if\ cdt\ addr_t\ else\ addr_e$. Checking the feasibility of each branch (cdt and $\neg cdt$) also includes making sure the attacker does not exceed their fault budget $m_{\mathcal{A}}$ to explore it. Hence the number of faults check ($nb_f \leq m_{\mathcal{A}}$) is appended to each satisfiability query. The number of faults check can be performed at different places. We found the best trade-off for forkless faults is to perform it at conditional jump evaluation, as checking it at the end of a path often involves exploring many unfeasible faulty paths.

Algorithm 4.3: Forkless conditional jump evaluation for data faults

Input: path predicate Φ, conditional jump instruction $if\ cdt\ addr_t\ else\ addr_e$
Data: fault counter nb_f, maximal number of faults max_f, worklist WL
Output: WL updated in place

```
1 Function eval_conditional_jump(Φ, cdt, addr_t, addr_e) is
2 |   if Φ ∧ cdt ∧ (nb_f ≤ max_f) is satisfiable then
3 |   |   Add (Φ ∧ cdt, addr_t) to WL
4 |   end
  |   /* Idem for else branch (¬cdt)                          */
5 end
```

Test Inversion Case. A test inversion fault happens when considering a conditional jump, leaving the standard SE process for assignments. We show in Algorithm 4.4 an expression containing the activation of the fault and its effect is computed by the function `FaultEncoding` similarly to data faults. The resulting faulty condition, $fault\_cdt$ is used in the rest of the conditional jump evaluation as the condition.

Algorithm 4.4: Forkless conditional jump evaluation for test inversions

Input: path predicate Φ, conditional jump instruction $if\ cdt\ addr_t\ else\ addr_e$
Data: fault counter nb_f, maximal number of faults max_f, worklist WL
Output: WL and nb_f updated in place

```
1 Function eval_conditional_jump(Φ, cdt, addr_t, addr_e) is
2 |   Φ', fault_cdt, nb_f := FaultEncoding(Φ, cdt, nb_f)
3 |   if Φ' ∧ fault_cdt ∧ (nb_f ≤ max_f) is satisfiable then
4 |   |   Add (Φ' ∧ fault_cdt, addr_t) to WL
5 |   end
  |   /* Idem for else branch (¬fault_cdt)                   */
6 end
```

Instruction Skip Case. This fault model changes each instruction handled by the SE to include the possibility of it having no effect and going to the next instruction in memory. It is hence implementation specific and we refer the reader to implementation details provided in Section 5.3.3.3. In particular for assignment evaluation, the process presented in Algorithm 4.2 is used.

Implementation Details. We present our implementation of those algorithmic modifications to a standard SE engine in Section 5.3 for binary-level analysis inside the BINSEC tool.

4.3.3.3 Properties

We now consider the properties of the FASE algorithm.

Proposition 4.2. *The FASE algorithm is correct and k-complete for adversarial reachability.*

Proof. If our algorithm finds an adversarial path reaching the target location l, by providing specific input values and a fault sequence, then an attacker executing the program with the provided inputs and performing the proposed faults will reach its goal. Our algorithm is based on symbolic execution with bounded path depth and explores all possible adversarial paths according to the considered attacker model, hence its k-completeness for adversarial reachability. □

FASE will propose a set of adversarial paths if there are some, it is not guaranteed to provide all possible adversarial paths with respect to fault placements.

Definition 4.5 (Path)**.** *We define a path by its control flow, that is to say, by the sequence of branch choices and jump addresses leading to the target location.*

Definition 4.6 (Equivalent Adversarial Paths)**.** *We define an adversarial path equivalence class by the set of adversarial paths that follow the same control flow.*

Proposition 4.3. *FASE will only report one adversarial path per equivalence class.*

Proof. FASE injects faults in a symbolic way, that is to say, they do not have concrete placements during the path exploration performed by the SE analysis. Only at the end of one path reaching the target location will the analysis ask an SMT solver for the satisfiability of the adversarial path and a model for symbolic values. Hence, FASE provides only one model (sequence for faults) for each path reaching the attacker's goal. □

Remarque 4.1. In a multiple fault analysis, FASE is not guaranteed to output minimal adversarial paths in the sense that there may be faults present that could be removed while the path still reaches the attacker's goal.

Proposition 4.4 (Tightness of FASE)**.** *Consider a single path with no branching instruction and a code location to reach at the end, together with f possible fault locations and a maximum of $m_{\mathcal{A}}$ faults. Then an SE injecting forking faults yields up to $C^f_{m_{\mathcal{A}}}$ [1] paths to analyze, and as many queries to send to the solver. In the same scenario, FASE (with forkless faults) will analyze only the original path, and send a single query to the solver.*

[1] Remind that k among n, written C^n_k is $C^n_k = \binom{k}{n} = \frac{n!}{k!(n-k)!}$

Remarque 4.2. The Forkless encoding increases query complexity, as described in Section 4.3.2.5 and experimentally verified in Section 6.3.2. We present in the remainder of this chapter two mitigation techniques.

4.4 Optimizations

As previously mentioned, FASE introduces many new symbolic variables which increase the burden of SMT solving in the analysis. FASE is based on a trade-off, no path explosion at the price of more complex queries. Our goal is to reduce this query complexity in an effort to get the best of both worlds.

In this section, we propose two optimizations, Early Detection of fault Saturation (EDS) and Injection On Demand (IOD), that aim to produce queries with fewer faulty expressions in them to alleviate the solver's efforts, while remaining correct and k-complete. The evaluation of each optimization's effectiveness is performed in Section 6.3.2.

4.4.1 Early Detection of Fault Saturation (EDS)

Our first optimization is called Early Detection of Fault Saturation, or EDS for short. The goal of EDS is to *stop fault injection as soon as possible*. The idea is to detect when the attacker has necessarily spent all their fault budget in the currently explored adversarial path, meaning they can't inject more in the rest of that path. Fewer faults injected in a path translates to fewer faulty expressions in the queries, hence, a reduced query complexity.

4.4.1.1 Supported Fault Models

The saturation check itself can be performed anywhere, at the price of more queries. We discuss here how it can be applied to our various fault models.

Data Faults. With forkless data faults in mind, we selected conditional jump evaluation as the saturation check location, taking advantage of the branch feasibility computation already performed. Checking it at each fault location wouldn't provide more information, and only checking at the end would be useless.

Test Inversion Faults. For this fault model, injection locations are already conditional jump evaluation. EDS can be applied straightforwardly, like for data faults. We do not believe EDS would provide much performance gain as the number of fault locations is very limited and the effect of test inversion fault does not add a new symbolic variable to be expressed, it is simply the opposite of the condition. Hence it would add queries and computation to try and mitigate a limited added query complexity. For those reasons, we believe that EDS could be used with test inversion faults, but with limited benefits. It has not been implemented in this thesis.

Instruction Skip Faults. We propose again to apply EDS at conditional jump evaluation, to take advantage of the information of branch feasibility. We believe EDS can be implemented for instruction skips. However, we made the choice in this thesis to implement for instruction skip faults only the best performing optimization, Injection On Demand, detailed in the next section. Section 6.3.2 shows the experimental results for data fault that we believe can be extended to instruction skip since assignments

are the most prevalent instructions in assembly. Hence, EDS for instruction skip faults has not been implemented in this thesis.

4.4.1.2 Algorithm

EDS modifications to FASE concern only the conditional jump evaluation, illustrated in Algorithm 4.5.

Algorithm 4.5: FASE-EDS conditional jump evaluation

Input: path predicate Φ, conditional jump instruction
$if\ cdt\ then\ addr_t\ else\ \ addr_e$
Data: fault counter nb_f, maximal number of faults max_f, worklist WL
Output: WL updated in place

```
1 Function eval_conditional_jump_EDS(Φ, cdt, addr_t, addr_e) is
2 |  if Φ ∧ cdt ∧ (nb_f < max_f) is satisfiable then
3 |  |  Add (Φ ∧ cdt, addr_t) to WL
4 |  else if Φ ∧ cdt ∧ (nb_f == max_f) is satisfiable then
5 |  |  Stop injection in this path
6 |  |  Add (Φ ∧ cdt, addr_t) to WL
7 |  end
  |  /* Idem for else branch (¬cdt)                                  */
8 end
```

Instead of checking whether a branch can be explored without exceeding the maximum number of faults ($nb_f \leq max_f$), we double the check:

1. First we check whether the branch can be explored with strictly fewer faults than allowed ($nb_f < max_f$). If the query is satisfiable, the analysis continues down that branch as usual;
2. If not satisfiable, we check whether the branch is feasible with exactly the maximal number of faults allowed ($nb_f == max_f$). If not, the branch is infeasible and we stop as usual. Yet, if it is feasible, the saturation detection is triggered and we know that we have spent all allowed faults. We can thus continue the exploration *without injecting any new fault* in the corresponding search sub-tree, leading to simpler subsequent queries.

4.4.1.3 Properties

We now consider the properties of FASE augmented with EDS, which we refer to as FASE-EDS.

Proposition 4.5. *FASE-EDS is correct and k-complete for the adversarial reachability problem.*

Proof. FASE-EDS remains correct as it does not modify the path predicate computation, and it remains k-complete as it only prunes fault injections that are actually infeasible – and would have been proven so by the solver, later in the solving process. □

4.4.2 Injection on Demand (IOD)

Our second optimization is called Injection On Demand, or IOD for short. The goal of IOD is to *inject fault on demand*, or as late as possible. The idea is to explore a path without injecting faults, then, only when they become needed to continue the exploration, they are added retrospectively to the path predicate. With this optimization, there are most of the time fewer faulty expressions in queries than what the attacker could do, since some injections have been delayed, hence reducing query complexity.

4.4.2.1 Supported Fault Models

We discuss here how it can be applied to our various fault models.

Data Faults. This optimization was designed with forkless data faults in mind. As they add complexity to the path predicate, it is particularly interesting to delay the injection until it is truly needed. This is evaluated at conditional jump locations to ensure all branches possible with faults are indeed explored.

Test Inversion Faults. Again, test inversion fault injection locations are already conditional jump evaluation. We do not believe IOD would provide much performance gain as the number of fault locations is very limited, only to conditional jump instructions, while data assignments are predominant, especially in assembly. Furthermore, the effect of test inversion fault does not add a new symbolic variable to be expressed, it is simply the opposite of the condition. Hence it would add queries and computation to try and mitigate a limited added query complexity. For those reasons, we believe that IOD could be used with test inversion faults, but with limited benefits. It has not been implemented in this thesis.

Instruction Skip Faults. This optimization works well with the data-related part of instruction skip, in a similar manner as for data faults. The algorithm retrospectively adding faults needs to be extended to take into account the other types of instructions. This mechanism is detailed in Section 5.3.5, at the implementation phase, when the types of instructions considered have been described.

4.4.2.2 Algorithm

IOD modifies FASE assignment evaluation as illustrated in Algorithm 4.6 and conditional jump as shown in Algorithm 4.7.

Dual Path Predicates. In order to perform the retroactive injection of faults when they are needed, we create a second path predicate.

- The first path predicate Φ is our main path predicate, upon which the analysis reasons and builds queries.
- The second path predicate Φ_F, or faulty path predicate, is our backup path predicate containing all the faults as the exploration progresses. It is used only when the analysis detects it needs more faults. It is then switched with the main path predicate, in effect, injecting faults retrospectively.

Assignment Process. The assignment evaluation is duplicated as shown in Algorithm 4.6:

- The normal symbolic assignment, with the original right-end-side expression $expr$, is performed in Φ;

– The fault injection is performed in Φ_F, which is updated with the fault encoding of the assignment, $expr'$.

Algorithm 4.6: FASE-IOD assignment evaluation

Input: path predicate Φ, faulted path predicate Φ_F, assignment instruction $x := expr$
Data: fault counter nb_f
Output: Updated Φ, Φ_F, nb_f updated in place

Function `eval_assign_IOD`$(\Phi,\ \Phi_F,\ cdt,\ x,\ expr)$ **is**
  $\Phi_F,\ expr',\ nb_f :=$ `FaultEncoding`$(\Phi_F,\ expr,\ nb_f)$
  return $(\Phi \wedge (x \triangleq expr),\ \Phi_F \wedge (x \triangleq expr'))$
end

Conditional Jump Process. The conditional jump evaluation is extended to test whether more faults are needed. Queries are built according to the following rules:

1. The first branch feasibility check is built with the main, simpler, path predicate Φ, encompassing the least number of faults. We continue this way as long as we can, meaning we rely on standard reachability as much as we can;
2. When encountering a branch infeasible with Φ, we then check whether this branch is feasible with all the possible faults seen so far, i.e. using Φ_F. If not, that is a stop, otherwise, we know that Φ does not encompass enough faults to go further. We then replace Φ by Φ_F (called a *switch*) at this stage, and thus continue with strictly more faults. The switch is straightforward as Φ_F and Φ only differ on fault injections. Then again, the new Φ will not accumulate any fault (until a new switch) while Φ_F continues accumulating all possible faults.

Algorithm 4.7: FASE-IOD conditional jump evaluation

Input: path predicate Φ, conditional jump instruction $if\ cdt\ then\ l_t\ else\ \ l_e$
Data: fault counter nb_f, maximal number of faults max_f, under approximation counter $under\_counter$, worklist WL
Output: WL updated in place

Function `eval_conditional_jump_IOD`$(\Phi,\ \Phi_F,\ cdt,\ l_t,\ l_e)$ **is**
  if $\Phi \wedge cdt \wedge (nb_f \leq max_f)$ *is satisfiable* **then**
    Add $(\Phi \wedge cdt,\ \Phi_F \wedge cdt,\ l_t)$ to WL
  else if $under\_counter \leq max_f$ **then**
    if $\Phi_F \wedge cdt \wedge (nb_f \leq max_f)$ *is satisfiable* **then**
      $\Phi := \Phi_F$
      $under\_counter := under\_counter + 1$
      Add $(\Phi \wedge cdt,\ \Phi_F \wedge cdt,\ l_t)$ to WL
    end
  end
  `/* Idem for else branch (`$\neg cdt$`) */`
end

Under-Approximation Counter. As a bonus, the number of path predicate switches gives us an under-approximation $under\_counter$ of the number of faults already needed

in the path under analysis. We use it to stop the injection early, when at least nb_f faults have been used.

4.4.2.3 Properties

We now consider the properties of FASE augmented with IOD, which we refer to as FASE-IOD.

Proposition 4.6. *FASE-IOD is correct and k-complete for the adversarial reachability problem.*

Proof. FASE-IOD explores the same feasible paths as FASE, hence preserving its properties. □

4.4.3 Combination of Optimizations

Since our two optimizations approach the problem of query complexity reduction from different angles, they can be combined, as illustrated in Algorithm 4.8. Taking FASE-IOD as a basis, saturation detection is added for the faulted path predicate Φ_F queries at the conditional branch evaluation. If the saturation is detected ($(nb_f \not< max_f) \wedge (nb_f == max_f)$), the main path predicate switches to Φ_F but for the last time along this path and Φ_F is not updated and queried upon anymore. This effectively stops fault injection, thus reducing subsequent query complexity.

Algorithm 4.8: FASE-IOD and FASE-EDS combination, conditional jump evaluation

Input: path predicate Φ, faulty path predicate Φ_F, conditional jump instruction $if\ cdt\ then\ addr_t\ else\ \ addr_e$
Data: fault counter nb_f, maximal number of faults max_f, under approximation counter $under\_counter$, worklist WL
Output: WL updated in place

```
1  Function eval_conditional_jump_EDS_IOD(Φ, Φ_F, cdt, addr_t, addr_e) is
2      if Φ ∧ cdt ∧ (nb_f ≤ max_f) is satisfiable then
3          Add (Φ ∧ cdt, Φ_F ∧ cdt, addr_t) to WL
4      else if under_counter ≤ max_f then
5          if Φ_F ∧ cdt ∧ (nb_f < max_f) is satisfiable then
6              Φ := Φ_F
7              under_counter := under_counter + 1
8              Add (Φ ∧ cdt, Φ_F ∧ cdt, addr_t) to WL
9          else if Φ_F ∧ cdt ∧ (nb_f == max_f) is satisfiable then
10             Φ := Φ_F
11             Stop switches and Φ' update and queries
12             Add (Φ ∧ cdt, Φ_F ∧ cdt, addr_t) to WL
13         end
14     end
       /* Idem for else branch (¬cdt)                                  */
15 end
```

Proposition 4.7. *The combination of FASE-EDS and FASE-IOD is correct and k-complete for the adversarial reachability problem.*

Proof. This combination also explores all possible paths for the considered attacker models, like FASE, hence preserving its properties. □

4.5 Discussion

In this section, we discuss the applicability of forkless fault encoding, from formalization to verification algorithm and its advantages for multi-fault analysis. We also consider using the forkless fault encoding to verify different properties and with different formal techniques. General techniques used in prior work in SWIFI have already been discussed in Section 3.4.

4.5.1 Fault Model Support - Formalization VS Algorithm

We discuss in this section what types of faults are supported by our formalization and those supported by our algorithm, FASE.

Our formalization is an extension of a transition system with adversarial transitions to represent the impact of an advanced attacker on a program. Our algorithmic approach based upon it, FASE, currently supports:

- Data faults involving a single instruction,
- Simple control-flow faults (test inversion and instruction skip).

While our formalization supports the following types of faults, our algorithm approach does not.

- Advanced data faults, whose effects span over multiple instructions. We believe that once characterized, those could be implemented by extending the program state with some form of counters to link the different effects together;
- Advanced control-flow faults such as arbitrary jumps. We have no algorithm more efficient in mind than a simple, and explosive, enumeration of possible effects for this class of fault models;
- Instruction corruption permanently changes an instruction, while we modify computation results. It is related to self-modification, which is a hard problem for symbolic execution. Supporting instruction corruption would require modeling instruction decoding and an efficient algorithm to avoid state explosion.

FASE can, in theory, support, or be extended to support, a vast number of different fault models. However, we do not guarantee our algorithm will be efficient for them, since they may involve new symbolic variables and an increased query complexity.

Representing Micro-Architectural Attacks. While Rowhammer's bit-flip effects are quite straightforward to model, modeling Spectre-type attacks is more difficult.

- For Spectre v1 (BTB, branch-target-buffer), the effect itself of mispredicting a branch can be modeled by a test inversion, however, the duration of the effect is limited in time and a rollback mechanism needs to be devised;
- In an LVI attack, transient execution is used to inject attacker-controlled values in the victim process. The effect of the attack is an arbitrary data fault, but with a limited effect duration to take into account;
- In Spectre v4 (STL, store-to-load), the processor speculates if a memory load depends on previous stores that are buffered, or independent and the value in

memory is the correct one. This can be seen as a restricted form of an LVI attack, where the attack effect is a data corruption dependent on previous values at the considered memory address. If those addresses are symbolic, the analysis can suffer a state explosion. Again, the effect is of limited duration.

While we believe FASE could be adapted to model temporary effects and state rollback, the application of our optimizations is not straightforward.

Other types of attacks exploit micro-architectural features and would require extending the program state with micro-architectural components [LBD+18, LBDPP19, TAC+22] to represent:

- permanent data faults,
- faults in the instruction prefetch buffer,
- faults in hidden registers,
- faults in the multiplication unit,
- faults in the forwarding mechanism,
- skip faults on variable-length instruction set [ACD+22] with miss-aligned instructions.

While FASE is architecture-independent, extending the program state as such would make the analysis architecture dependent, and would have to be adapted for each architecture considered.

Probabilistic Faults. Some techniques [DPdC+15], mostly simulation-based, implement probabilistic fault models to account for the plausibility of a fault, depending on injection parameters. We believe our formalism could be extended to support probabilistic adversarial transitions. However, our current algorithm cannot support those faults without a path explosion, forking to consider each fault having a non-zero probability.

Related Work. Fault model support of other SWIFI works is presented in Section 4.6.1.

4.5.2 Forkless Faults and Multi-Fault Analysis

We now discuss the interest of the forkless fault encoding for a multi-fault analysis.

The goal of the forkless encoding is to be able to represent attackers able to perform multiple faults in one attack while preventing the path explosion faced by existing techniques. We do this by embedding the possibility of a fault at each possible location without forking the path each time, at the price of more complexity in the path predicate, which makes for more complex SMT queries, the bottleneck of symbolic execution. SWIFI is a trade-off between exploring more paths or having more complex queries. We chose to explore the side of more complex queries. Experiments (see Chapter 6) will show that our approach yields the best performance in most cases, even without our optimizations reducing the number of injection points in queries.

While our algorithm is abstraction-level independent, we implement it in Chapter 5 for a binary-level analysis, which is harder than source-level analyses (see Section 3.3).

4.5.3 Forkless Encoding for Other Properties

The forkless encoding can surely benefit other classes of properties to be achieved by the attacker, especially those known to be supported by (extensions of) symbolic execution, for example:

- Trace properties that can be evaluated based on a transition system, such as use-after-free, could also be evaluated under attack when evaluated based on our adversarial transition system instead. This corresponds to considering an attack goal other than adversarial reachability, with a more complex oracle. In particular, an extension to other trace properties allows to consider complex multi-step attacks where a vulnerability is used to trigger the next one;
- K-hyperreachability properties (secret leakage, privacy leakage, violation of constant-time, etc.) [DBR20] are often evaluated with relational symbolic execution, where two traces following the same path are computed and then compared. Faults can be added to that computation similarly to how we included them in symbolic execution. This could be used to extend attacker goals beyond reachability, in particular towards information leakage, or serve as a guide for multi-step attacks;
- The recent robust reachability proposal [GFB21] and its quantified version [BG22] consider the control an attacker needs, or not, over each program parameter to trigger a bug. This work aims to mitigate symbolic execution false positives and find highly replicable bugs. This technique and ours could be combined to quantify the control an attacker needs over the injected fault. For instance, does the attacker have to corrupt a value to 0xdeadbeef as the adversarial symbolic execution says, or any random value would work? In particular, this could help an attacker decide what injection mean provides an adequate level of control over the injected fault to perform the attack. However, robust reachability requires a more expressive, and harder-to-solve, SMT theory than we use. We believe the combination of the two techniques would suffer scalability issues.

4.5.4 Forkless Encoding for Other Formal Methods

While in this thesis, we focus on symbolic execution, we believe the main optimization ideas developed here can be used with other formal techniques, e.g. Bounded Model Checking, Abstract Interpretation or CEGAR model checking. Note that for each of them, fault injection may result either in path explosion or precision loss. Still, our forkless encoding should be able to help at least all approaches based to some extent on path unrolling.

Previous works have explored non-forking injection techniques with abstract interpretation [LFBP21], deductive verification [MKP22] and bounded model checking [TAC+22].

Lacombe *et al.* [LFBP21] use abstract interpretation, through the Frama-C [KKP+15] plugin Eva [BBY17]. They propose a form of forkless faults, xoring the result of a computation with a free variable, manually added to a C program. However, they rely on abstract interpretation only to filter injection points, not to find adversarial paths directly.

Martin *et al.* [MKP22] use deductive verification at C level through the Frama-C plugin LTEST [BCDK14] with test inversion faults. They use an uninterpreted function to simulate the activation of the fault and its behavior. Only the contract of this function is written, leaving Frama-C to verify if the security property can be violated.

Tollec *et al.* [TAC+22] leverage bounded model checking for fault injection analysis, except they reason at the RTL level, modeling the CPU micro-architecture. The RTL model takes the place of the program, whose input is the binary program analyzed, with possible faults on each wire on the RTL model. The analysis can be considered

completely forkless as the possibility of faults and their effect are encoded directly into the model of the bounded model checker, which never forks.

4.5.5 Forkless Encoding and Instrumentation

Several prior works use code-level instrumentation [MKP22] or llvm level instrumentation [PMPD14, LFBP21, LHGD18] in order to leverage standard program analyzers as is. The forkless encoding we propose can also be used this way, for more flexibility. However, since using code-level instrumentation takes a step back from the symbolic execution engine, not all optimizations still make sense. While we believe Early Detection of Fault Saturation could be adapted, Injection On Demand requires heavy path predicate transformation that does not seem feasible in code instrumentation. Actually, we performed some experiments with Klee and C-level forkless instrumentation in Section 6.5, and do observe significant performance improvement compared to a forking instrumentation.

4.6 Related Work

In this section, we present the related work regarding fault model support and multi-fault analysis. We also explore other fields considering attackers and other formalisms that have been proposed.

4.6.1 Fault Model Support

We list in Table 4.3 the main SWIFI works and the fault models they support. Most only support one or very few fault models that are typically simple data faults or control-flow faults (test inversion and instruction skip). To our knowledge, only one [TAC+22] includes micro-architecture details in their fault injection analysis which allows them to reason about complex faults.

4.6.2 Multiple Fault Analysis

We now present how existing techniques support multi-fault in their analyses. Among SWIFI techniques, half rely on the mutant approach [CCG13, RG14, GWJLL17, CDFG18, GWJL20], which is not designed for multi-fault analysis has the number of mutants explodes quickly. The other half relies on forking techniques [PMPD14, BBC+14, BHE+19, LFBP21, Lan22] whose scalability in the number of faults considered is hindered by path explosion. Very few works consider multi-faults [PMPD14, LFBP21, MKP22, Lan22, GHHR23] (see Table 4.3).

Potet *et al.* [PMPD14] only considers test inversion faults, which are much less frequent than data faults. They use a preprocessing consisting of a basic block coloring algorithm based on reachability in a graph to filter necessary fault locations, locations where faults should be avoided and uncertain locations. That information is then embedded in a meta-mutant which is analyzed by symbolic execution with KLEE [CDE+08]. Hence, the only forking faults handled by the symbolic engine are uncertain fault locations of a fault model having few in total.

Lacombe *et al.* [LFBP21] use abstract interpretation, through the Frama-C [KKP+15] plugin Eva [BBY17], to perform a data dependancy analysis reducing injection points

Table 4.3: Fault model support

Reference	Analysis level	Arbitrary data	Bit-flip	Other data fault	Test inversion	Instruction skip	Other	Multi-fault
Christofi [CCG13]	C	✗	✗	✓	✗	✗	✗	✗
LAZART [PMPD14, LFBP21]	llvm	✓	✗	✗	✓	✗	✗	✓
Rauzy [RG14]	Custom	✓	✗	✓	✗	✗	✗	✗
EFS [BBC+14]	Binary	✗	✗	✗	✗	✓	✗	✗
Given-Wilson [GWJLL17]	ISA	✗	✗	✓	✓	✓	✗	✗
Carré [CDFG18]	Binary	✗	✓	✗	✗	✗	✗	✗
RobustB [BHE+19]	Binary	✓	✗	✗	✗	✓	✗	✗
Given-Wilson [GWJL20]	Binary	✗	✓	✓	✗	✓	✗	✗
Lacombe [LFBP21]	C/llvm	✓	✗	✗	✗	✗	✗	✓
Martin [MKP22]	C	✗	✗	✗	✓	✗	✗	✓
Tollec [TAC+22]	RTL	✗	✗	✗	✗	✗	✓	✗
Lancia [Lan22]	ISA	✓	✓	✓	✓	✗	✗	✓
SAMVA [GHHR23]	ISA	✗	✗	✗	✗	✓	✗	✓
FASE	ISA	✓	✓	✓	✓	✓	✗	✓

on C programs. The remaining injection points are then provided to a more recent version of Lazart [PMPD14], able to reason on data faults, to compute adversarial paths. Their fault injection pruning technique based on data-flow analysis is complementary to our own method – still, static analysis at binary level is known to be hard.

Martin *et al.* [MKP22] use an uninterpreted function whose contract embeds the activation of test inversion faults and a counter to limit them to the attacker's budget. This technique discharges the multi-fault reasoning to a solver, similar to our approach, except that we have a concrete fault encoding, allowing us to reason about and prune fault injection locations. Furthermore, they reason at the C level.

Gicquel *et al.* [GHHR23] have developed an injection technique based on graph reasoning for the instruction skip fault model at the binary level. All possible faults are encoded by adding edges on the graph representing the program with basic blocks, each labeled as 'neutral', 'skip' or 'execute'. Their algorithm then builds attack paths placing instructions skips of arbitrary width to cover all 'skip' nodes while avoiding all the 'execute' nodes. Here, increasing the attacker fault budget doesn't change the extended graph, it only offers more freedom to the placing algorithm.

In summary, those techniques are able to perform multiple fault analyses by:

- using preprocessing filtering fault locations to limit the path explosion [PMPD14, LFBP21];
- reasoning at a higher abstraction level than ISA, such as C [LFBP21, MKP22] or llvm [PMPD14];
- fine-tuning their multi-fault technique to a single fault model [PMPD14, MKP22, GHHR23];
- limiting multi-fault considerations to two faults in practice [LFBP21, MKP22, Lan22].

4.6.3 The Attacker in Different Security Fields

In this section, we overview how attackers are represented in various fields for security analysis.

Protocol Verification. It is common in the field of automated formal verification of cryptographic protocols to consider models of attackers, typically extensions of the "Dolev-Yao" model [DY83]. In the Dolev-Yao model, the attacker can see, intercept and generate messages exchanged over the network. Cryptographic primitives are represented as abstract operators. The attacker only knows what has been previously exchanged [AC04] to try and guess the secret key. Conversely, attackers can also be represented by what they cannot do [BCL14].

Control-Flow Integrity. In software security, control-flow integrity attacks have been categorized by the capability an attacker needs [BCN+17] to perform an attack or against which a countermeasure will hold. Those capabilities are of the form 'write something anywhere', 'write anything anywhere', etc. These efforts have been restricted to manual reasoning.

Bug Triaging. With the advent of massive fuzzer usage, the number of bugs discovered in programs becomes much higher than the capabilities of developers to fix them. A rising area of research is bug triaging, where the severity and impact of bugs are evaluated to prioritize the most dangerous ones. Bugs are represented by the capabilities [JGH+22] they provide to the attacker willing to exploit them, such as the number of controlled bits.

High-Level Attacker. Some works intend to model an actual attacker, a "hacker" in action [CTB+17, ASA+15, BCR+19, CTB+19]. They aim to categorize high-level attackers, understand what capabilities they have or not, and when and why they favor one over another in the course of an attack. Those capabilities are of the form 'tamper with the execution environment', 'deobfuscate code', 'brute force attack', etc. While at a very high-level (workflow of the attack), these efforts are indeed relevant to better understand which capabilities should be granted to a low-level formal model of an attacker.

Precise Hardware Fault Models. Determining the most appropriate fault models representing hardware faults is an ongoing research question. Fault models [GMA22, RBSG22] can be derived from experiments. Methodologies have been proposed [DPdC+15]. However, software fault models miss architecture-specific behavior related to control signals in the pipeline, able to bypass countermeasures [LBD+18]. Faults on data are independent of hardware, and faults on instructions depend on the Instruction Set Architecture (ISA).

4.6.4 Extending Existing Formalisms

We highlight the main works that extended existing formalisms to include the capabilities of an attacker.

Adversarial Logic. Here, Vanegue [Van22] extends incorrectness logic [O'H19], propagating accumulated errors to determine a bug exploitability. This bug-finding (under-approximation) technique integrates an attacker from the side-channel and information knowledge point of view. As our technique does not cover information leakage, it would be interesting to try and combine both approaches in order to represent a more generic

attacker, able to also seek knowledge.

Operational Semantic. Given-Wilson et al. [GWL20] propose a formalization of fault injection on a program using Turing machines. The attacker model considered can inject active faults without consideration of the injection mean and is very generic in supported faulty behaviors. This work is orthogonal to ours, however, to our knowledge, no algorithm has been built for this formalization. Also, Fournet et al. [FR08] propose a type system for program-level non-interference, taking into account an active adversary modeled as adversarial components able to perform any action at certain steps of the program.

4.7 Conclusion

Summary. In this Chapter, we started by describing how we modeled an advanced attacker, an entity with attack capabilities, a limitation on the number that can be used and an attack goal expressed as a program location to reach. We then showed how to extend reachability to include adversarial transitions, building the new concept of adversarial reachability. Finally, we extended the symbolic execution algorithm, traditionally verifying reachability, to include attacker actions as fault injections, building adversarial symbolic execution (ASE). ASE is correct and k-complete for adversarial reachability. The main limitation of existing SWIFI techniques is the path explosion experienced. To prevent it, we proposed a new forkless fault encoding, that has the drawback of increasing query complexity. We designed two optimizations dedicated to reducing this added complexity.

Extension to Published Work. Compared to the ESOP 2023 [DBP23] paper, we added the instruction skip fault model and adapted our optimizations for it. We also proposed several discussions, in particular regarding the expressivity of our adversarial transition system.

Coming Next. In the next chapter, we present our implementation of ASE into a tool, BINSEC/ASE, to verify in practice the vulnerability of a program to an attacker model. Its practical interest and effectiveness are investigated in Chapter 6.

Chapter 5

The BINSEC/ASE Prototype

Contents

In this chapter, we present BINSEC/ASE, our software-implemented fault injection tool, based on Adversarial Symbolic Execution (described in Chapter 4). After a brief introduction explaining the goal and interest of our approach (Section 5.1), we describe the BINSEC tool[1] (Section 5.2) and how we modified it to build BINSEC/ASE (Section 5.3). Then, we provide a user guide detailing a methodology to add a new benchmark (Section 5.4), as well as a developer guide showcasing the methodology to add a new fault model through the instruction skip example (Section 5.5). At the end of this chapter, we discuss the tool's particularities and limitations (Section 5.6). The evaluation of BINSEC/ASE is detailed in Chapter 6.

The BINSEC/ASE tool can be found as an artifact on GitHub[2] and Zenodo[3].

[1] BINSEC version of September 1^{st} 2021
[2] https://github.com/binsec/esop2023_artefact
[3] https://zenodo.org/record/7507112

5.1 Overview

Adversarial Reachability aims at reasoning about the impact of an advanced attacker, able to inject multiple faults, on a program's security properties. We designed an algorithm to answer the Adversarial Reachability problem, Adversarial Symbolic Execution. We implement it inside a tool, BINSEC/ASE, answering Adversarial Reachability from a bug-finding point of view (under-approximation).

This tool's aim is to be integrated into the security evaluation of programs. Once a risk assessment has identified threat models and security properties (not in the scope of this thesis), BINSEC/ASE can help the security expert determine if the program is vulnerable or not, and provide attack paths allowing the attacker to violate the security property. This can be used to identify critical code sections to try and mount full attacks on the program. Another use is countermeasure evaluation. Adding protections also adds attack surface, so protecting a program against one attack may open the way for a new attack for the same attacker model, hence an iterative process using a tool such as BINSEC/ASE.

In the following, we motivate the main high-level design choices made for BINSEC/ASE design.

- **ISA Abstraction Level:** we implemented Adversarial Symbolic Execution (ASE) at the ISA level. ISA stands for Instruction Set Architecture, also called assembly, it defines an interface between hardware and software. An analysis at ISA level makes the tool independent from source-level languages and constructs hindering analysis precision such as register dynamics. We also wanted to be independent of the specific micro-architecture of the system running the program, which requires heavy modeling for each target system, hindering scalability and for which specifications are often closed-sourced. A program may be launched on multiple target architectures, requiring multiple costly analyses when considering the micro-architecture. ISA represents directly the code executed by the processor, instructions and operands. Many fault models characterizing physical fault injection in the literature are at the ISA level;
- **Building Inside BINSEC:** BINSEC is a symbolic execution engine working at the ISA level, designed and developed at CEA LIST, which makes help to understand BINSEC's inner workings and to debug more easily available, which saves a lot of time and effort. The BINSEC tool consists of about 60k loc of Ocaml, the symbolic execution engine contains around 3.2k loc. We modified only the SE engine, implementing ASE in 5.8k loc;
- **ISA Support:** we currently support Intel x86-32 bits and ARM 32 bits architectures. Support can be extended to other architectures as they become available in BINSEC.

5.2 Background: the BINSEC Tool

In this section, we present the BINSEC tool, upon which we built BINSEC/ASE. Details of our modifications can be found in the next Section (5.3). BINSEC is an open-source symbolic execution engine for binary programs developed at the CEA by the Binsec team[4]. We use BINSEC version 0.4.0 in this work, forked September 1st 2021.

[4] https://binsec.github.io/

5.2.1 BINSEC Presentation

We implement BINSEC/ASE on top of the BINSEC symbolic engine [DBT+16, DB15, BHL+11]. It has already been used in a number of significant case studies [BDM17, RBB+19, RBB+21, DBR20, DBR21], and it is notably able to achieve bounded verification (k-completeness) and to reasonably deal with symbolic pointers [FDBL18].

5.2.2 General Work-Flow

We detail in this section the part of BINSEC's general workflow that is relevant to the work presented in this thesis, from user inputs, internal path exploration and communications with the SMT solver, to the provided output. Figure 5.1 provide an overview of this workflow.

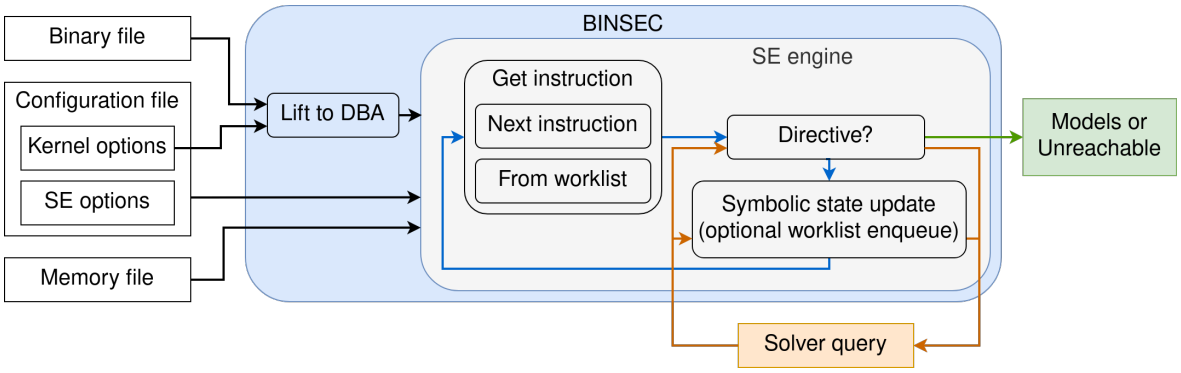


Figure 5.1: Overview of BINSEC workflow for symbolic execution

5.2.2.1 Parameters

BINSEC can be parameterized using command line options or by providing it with a configuration file. The user defines a number of exploration parameters. We list here the main ones we use, divided into two categories, the kernel options and the symbolic exploration options.

Kernel options. They define the core elements required for BINSEC to run:
- the path to the binary *file* to analyse,
- the *ISA* for which the binary has been compiled,
- the *entrypoint* from which the symbolic analysis starts. It can be an address or more commonly the symbol of a function like <*main*>.

Symbolic execution options. They describe SE parameters and goals.
- Exploration objectives are defined in terms of *directives*. They can be of various forms such as *reach* an address once or multiple times, with or without a condition, *cut* the exploration when reaching an address, with or without a condition, or force constraints at a location with an *assume*;
- BINSEC performs an analysis bounded by the maximal number of instructions analyzed per path, set by the user;
- As the analysis starts at the defined entrypoint, the user can specify some variable initializations in a *memory* file. In particular, it is used to provide an initial address to the stack pointer, easing the analysis without loss of generality.

5.2.2.2 Intermediate Representation: DBA

To provide an analysis independent of the specific ISA used, BINSEC starts by reading the binary and lifting the ISA instructions to an intermediate representation called DBA (short for Dynamic Bitvector Automata). This is a language specific to BINSEC. It abstracts each ISA instruction into a block of DBA instructions detailing registers, memory and flag updates with arithmetic expressions, as well as the next instruction to execute. There are four main kinds of DBA instructions: assignments, conditional jumps, static jumps and dynamic jumps.

Table 5.1 illustrates the DBA syntax from a very simple C example, with the associated ISA instructions (Intel x86). For readability, the updated expressions of flags are not displayed, except when relevant. The ISA instructions don't reflect the full expressivity of the *assume* C statement, just the conditional branching.

5.2.2.3 Analysis Workflow

The symbolic execution engine will start its analysis at the *entrypoint* defined by the user with the initialization contained in the provided *memory file*. Then instructions are decoded one by one in their execution order.

- In the case of assignments, updates are made to the symbolic state, which records the current value, concrete or symbolic, of each variable;
- For a conditional jump, the condition is added to the path predicate constraints sent to the SMT solver. The analysis will fork if both branches are possible, selecting one to keep exploring and storing the other in a worklist to continue the exploration later;
- In the case of a static jump, the analysis will decode the instruction at the newly provided address and continue from there;
- Lastly, for dynamic jumps, also known as indirect jumps (the target address has been computed at runtime and is stored in a register), the analysis asks the SMT solver for n possible values (3 by default but it can be changed through configuration) for the jump target address and forks the analysis, continuing one and storing all the others in the worklist. In practice, only one jump address usually makes sense.

BINSEC uses a depth-first approach with its worklist of paths to analyze. The exploration of a path ends when it meets a directive's criteria or when it exceeds the maximal instruction depth.

5.2.2.4 SMT Solvers

BINSEC uses the QF_ABV theory, standing for Quantifier Free, Arrays and Bit-Vectors, to discharge created formulas. BINSEC has a native binding with the SMT solver Bitwuzla[5] [NP20], winner of SMT-COMP 2022[6] for its category. It also interfaces with other solvers, such as z3[7] [DMB08] or Boolector[8] [BB09], through the use of the smtlib, a universal API for SMT solvers. A connection through the smtlib is slower than a native binding, internally optimized for the solver.

[5] https://bitwuzla.github.io/
[6] https://smt-comp.github.io/2022/results/qf-bitvec-single-query
[7] https://github.com/Z3Prover/z3
[8] https://boolector.github.io/

Table 5.1: Example of C and Intel x86 instructions translated to DBA

C code	Intel x86	DBA
`int x = 9;`	`mov [ebp + 0xfffffff4], 0x9`	`0: @[(ebp<32> + -12<32>),4] := 9<32>;` `1: goto (08049d1c, 0)`

`x = x + 1;`	`add [ebp + 0xfffffff4], 0x1`	`0: res32<32> := (@[(ebp<32> + -12<32>),4] + 1<32>);`
		`1: OF<1> := [...];`
		`2: SF<1> := [...];`
		`3: ZF<1> := [...];`
		`4: AF<1> := [...];`
		`5: PF<1> := [...];`
		`6: CF<1> := [...];`
		`7: @[(ebp<32> + -12<32>),4] := res32<32>;`
		`8: goto (08049d20, 0)`

`assert(x == 8);`	`cmp [ebp + 0xfffffff4], 0x8`	`0: res32<32> := (@[(ebp<32> + -12<32>),4] - 8<32>);`
		`1: OF<1> := [...];`
		`2: SF<1> := [...];`
		`3: ZF<1> := (0<32> = res32<32>);`
		`4: AF<1> := [...];`
		`5: PF<1> := [...];`
		`6: CF<1> := (@[(ebp<32> + -12<32>),4] <u 8<32>);`
		`7: goto (08049d24, 0)`

	`jz 0x8049d44`	`0: if ZF<1> goto (08049d44, 0) else goto 1` `1: goto (08049d26, 0)`

5.2.2.5 Output

Among the usages BINSEC has, we are interested in standard reachability in this thesis. When the SE engine encounters a code address associated with a reach directive, it creates the associated formulas for the SMT solver to assess satisfiability. The solver will either answer that the path does not validate the directive and BINSEC forwards the information that the code location is unreachable from that path, either the solver provides a model for symbolic variables such that the formula is true and BINSEC forwards this model for symbolic inputs to the user.

5.2.3 Summary

BINSEC is a symbolic execution engine for binary programs with good properties:

- It can assess the standard reachability of code locations;
- It is correct assuming correct logical encoding of path predicates and correctness of the underlying solver (if the solver answers SAT with a model, then the formula is indeed satisfiable and the model is a real one);
- It achieves bounded verification (k-completeness), assuming all dynamic jumps can be enumerated, which is usually the case in practice.

5.3 BINSEC/ASE Implementation

The work described in the rest of this chapter has been conducted during this thesis. BINSEC symbolic engine core is the only part we changed in order to implement BINSEC/ASE. We start by describing the main modifications made to BINSEC, before detailing the fault models' implementation and the implementation of our optimizations.

5.3.1 BINSEC/ASE Overview

From a user point of view, using BINSEC/ASE is very similar to using the BINSEC tool. The main difference is the attacker model the user provides in the configuration. BINSEC/ASE then assess the adversarial reachability of the attacker's goal location and outputs possible attack paths if they exist. Internally, several changes are needed in the symbolic execution engine to account for the attacker model's impact on the program. Figure 5.2 presents an overview of the BINSEC/ASE workflow, detailed in the following sections.

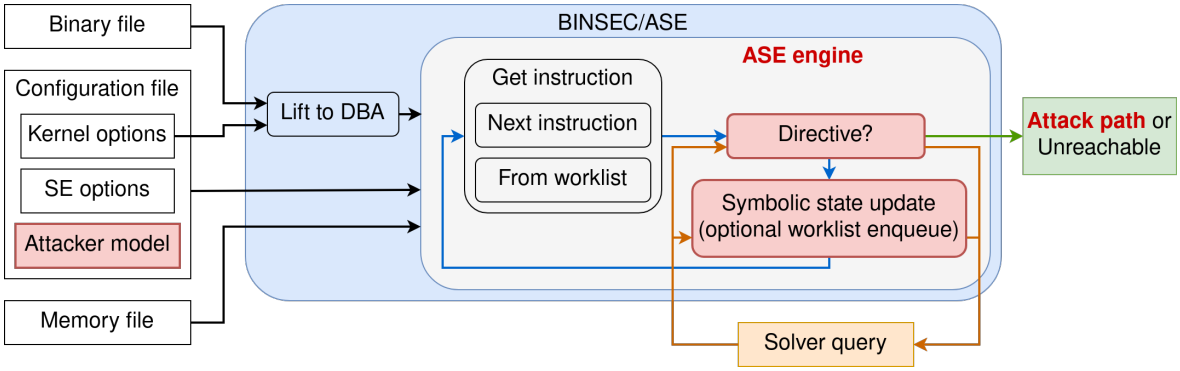


Figure 5.2: Overview of BINSEC/ASE workflow

5.3.2 ASE Implementation

The main modifications to the SE algorithm are described in chapter 4. We detail in this section practical details regarding their implementations.

5.3.2.1 Attacker Model Parameters and Goal

The user configures the attacker model to consider for the analysis. It takes the form of new parameters inside the configuration file, in particular, to specify the attacker's capability and the maximum number of faults that are allowed in one attack path. More details on the configuration of the attacker model are available in the user guide provided in Section 5.4.2.2. The attacker goal is expressed with a dedicated *reach* directive.

5.3.2.2 Global Variables

We introduce new global variables in the symbolic state to keep track of injected faults.

Fault Counter. We use a symbolic variable, called *FAULT_NUMBER* to keep track of how many faults are active, computed by summing the individual fault activation variables. It is initialized at 0 at the initialization of the analysis.

Fault Budget. We recover the configuration value given by the user setting the maximum number of faults allowed to the attacker model and store it inside a symbolic variable called *MAX_FAULTS* (with a concrete fixed value) at the analysis initialization. Having the fault budget as a symbolic variable makes it easy to compare it to our symbolic fault counter in queries.

Activation Constraint. Without some additional constraints, a fault could be declared active but have no effect. For instance, a value could be corrupted to a new but identical value, or a reset fault could be performed on a variable already at 0. To count only active faults, truly modifying a variable's value, we generate and aggregate a set of constraints in a symbolic variable called *ACTIVATION_CONSTRAINTS*. This variable is a boolean value initialized at true. For each fault in the path predicate, *ACTIVATION_CONSTRAINTS* is updated by the computation of a logical *and* of its current value and the new constraint with respect to the fault model used. A constraint ensuring *ACTIVATION_CONSTRAINTS* is true is added to each query sent to the SMT solver. To sum up, we add activation constraints to remove ineffective faults for a more accurate fault count and possibly help the solver eliminate them.

5.3.2.3 Assignments

In the program analysis process, when an assignment instruction is encountered, the symbolic state update is augmented.

Behavior Wrapper. We created a wrapper around the assignment behavior, which is in charge of deciding which behavior should be applied for this assignment. In particular, should the normal assignment behavior happen, or a behavior injecting a fault?

Fault Filter. Each assignment isn't a possible fault location, there are several conditions to satisfy.

1. The user has to have selected an attacker model able to perform faults, i.e. the selected attacker capability isn't *None* and the fault budget is strictly greater than 0;

2. The instruction must be in the target ranges of the injection set by the user in the configuration;
3. The type of the variable on the left-hand side of the assignment cannot be a temporary variable (for DBA internal use with no architectural reality). Flags are faulted or not according to the attacker model;
4. As we have no efficient algorithm for faults on addresses, we set a threshold above which a variable is considered an address and is not faulted. This threshold is set by default at $0x05000000$ and can be changed by the user;
5. There are some variables the user may not want to fault, given in the configuration. If the assigned variable is in this *blacklist*, it is not faulted. We mainly use this feature to avoid faulting the stack pointer.

Fault Selection. Once the instruction qualifies for fault injection, a match-case selects the assignment function implementing the desired behavior. If it isn't a target for fault injection, the normal assignment function (from BINSEC) is selected. Those functions update the symbolic state according to the instruction and fault model.

5.3.2.4 Conditional Jumps

Conditional jumps can be a place of fault injection and bear some fault injection machinery.

Behavior Wrapper. We also created a wrapper around the conditional jump behavior, which is in charge of deciding which behavior should be applied for this conditional jump.

Number of Fault Check. A question was: when to make sure the number of active faults doesn't exceed the fault budget, for Forkless faults in particular? Checking the number of faults at each fault location is costly and doesn't bring more information, since all assignments done after the last jump don't bring new constraints. Checking the number of faults at the end of the path leaves room to explore unfeasible paths, that is to say, paths that require more faults than the user gave to the attacker model. We opted for the third option, checking the number of faults at each conditional jump. It takes advantage of the already existing queries at conditional jump locations, adds information due to the activation constraint and allows to prune infeasible paths. To implement this, we augment the normal conditional jump queries (one for the then branch and one for the else branch) with a boolean condition stating that the current value of the fault counter should be lesser or equal to the fault budget. We coupled this number of faults check with checking the validity of the activation constraints.

Fault Filter and Selection. BINSEC/ASE implements the test inversion fault model, which injects faults into conditional jumps. A filter is first applied, containing only elements 1,2 and 5 of the enumeration of the assignment fault filter (Section 5.3.2.3). Then, if the instruction is eligible for fault injection, the wrapper selects the faulty behavior, otherwise, it selects the normal behavior, always passing the augmented query.

5.3.2.5 Exploration Directives

The goal of the exploration is to assess the adversarial reachability of a code location. Our modifications to BINSEC extend its original reachability assessment to adversarial reachability.

On top of the fault injections, a guard is added to the reachability query, when assessing the feasibility of a path reaching a code location specified by a directive. It consists of a combination of the fault check (the current value of the fault counter needs to be lesser or equal to the fault budget) and the activation constraints check (ensuring ineffective faults are not activated).

5.3.2.6 BINSEC/ASE Output

BINSEC/ASE output is similar to BINSEC's. If there exists a model for symbolic variables such that the attacker's goal is reached, it will be displayed. While in BINSEC symbolic variables are mostly input of the program, with BINSEC/ASE, the model also includes an activation value for each possible fault location, and a symbolic corruption value depending on the fault model. For instance, there will be none for reset faults, it indicates which bit is flipped for a bit-flip fault model, or an exact value in the case of arbitrary data faults. More details are discussed in Section 5.3.3. A model for program input and a fault sequence defines an adversarial path.

5.3.2.7 Statistics

We extended BINSEC's computed statistics with, for instance, the number of injection points in total and per path, statistics on queries such as the queries trivially true or false, how many were created and how many were sent. We also implemented the optional computation of the number of *ite* operators per query, giving a sense of the query complexity. This is optional as it is costly to compute, requiring to traverse each query. Those statistics help with the development effort, in particular, to ensure the correctness of the analysis and assist the debugging process. They also allow comparing techniques, encoding, optimizations, etc. More information on their usage is available in the user guide (Section 5.4.4).

5.3.2.8 Implementation Details

BINSEC/ASE features other implementation details. They are not purely derived from Adversarial Symbolic Execution algorithms but are tweaks to improve performance.

Symbolic default value. When an evaluation is performed on a model, to see if it can be trivially resolved or if the solver needs to be used, and that model contains free, unconstrained variables, BINSEC attributes them an arbitrary value. It helps for performance inside BINSEC. However, since most of our symbolic variables relate to faults, we noticed it improves performance to give them the default value of 0. It means faults are not activated by default.

Simplification rules. The operator we favor for forkless encodings, the *ite* operator is rarely used in DBA natively. For this reason, very few algorithmic simplification rules for symbolic terms dedicated to the *ite* operator were implemented in BINSEC. Simplification rules are used when symbolic terms are created and before they are added to the symbolic state, path predicate or queried upon. This saves solver effort, which is the main bottleneck in standard symbolic execution. To accommodate our use of *ite*s, we implemented in BINSEC/ASE additional rules that have been backported to the BINSEC tool.

- When a unary operator, *unop*, *not* for instance, is applied to an *ite*, it can be forwarded to the then and else members of the *ite*;

- When a binary operator, *binop*, *plus* or *minus* for instance, is applied to an *ite* term and a constant, the binary operation can be forwarded inside the then and else members of the *ite*;
- When a test operation, *compop*, equal or lower than for example, is applied to an *ite*, the expression can be simplified if we know the results of the comparison for each internal member t and e of the *ite*.

Formal rules are presented in Table 5.2.

Original expression	Expression after simplification
unop (ite c ? t : e)	ite c ? (*unop* t) : (*unop* e)
binop (ite c ? t : e) cst	ite c ? (*binop* t cst) : (*binop* e cst)
binop cst (ite c ? t : e)	ite c ? (*binop* cst t) : (*binop* cst e)

Original expression	Condition	Simplification
compop (ite c ? t : e) cst	(*compop* t cst) && (*compop* e cst)	1
compop (ite c ? t : e) cst	¬(*compop* t cst) && (*compop* e cst)	¬c
compop (ite c ? t : e) cst	(*compop* t cst) && ¬(*compop* e cst)	c
compop (ite c ? t : e) cst	¬(*compop* t cst) && ¬(*compop* e cst)	0

Table 5.2: Simplification rules added to BINSEC/ASE

5.3.3 Forkless Fault Models

We detail in this section how our forkless fault models are implemented. To date, we have implemented arbitrary data faults, reset faults, bit-flips, test inversions and instruction skips. Those are generic and commonly used fault models to represent faults from hardware fault injection attacks and can be applied to other fault injection means.

5.3.3.1 Data Faults

Once this fault behavior has been selected for an assignment, it is possible to perform the number of faults check and the activation constraints check, though it is not the recommended way. If the checks don't pass, the normal –non faulted behavior is executed.

Otherwise, two new symbolic variables are created, one boolean value representing the activation of the fault, and a variable the size of the left-hand side variable representing the fault effect. As a convenience, we add a suffix to those variables' names with the address of the current instruction. It helps to analyze the results and locate the faults in the ISA of the program. The fault counter is incremented with the symbolic value of the activation variable.

All elements required to generate the activation constraint of the fault according to the fault model are set. We update the symbolic state with this additional constraint.

Finally, the new right-hand side expression is computed according to the fault model and returned to the wrapping function which performs the final symbolic state update.

A summary of this process is shown in Algorithm 5.1 for arbitrary data faults. Note that := denotes an assignment of code variables, while $\triangleq$ denotes the mapping of a variable in the symbolic state.

Algorithm 5.1: Implementation sketch of a forkless arbitrary data fault (with ite operator) assignment process

Input: path predicate Φ, assignment instruction $x := y$
Data: fault counter $NUMBER\_FAULT$ and activation constraint $ACTIVATION\_CONSTRAINT$ stored in Φ
Output: Updated Φ

```
1  Function eval_assign_normal(Φ, x, y) is
2  |  return Φ ∧ (x ≜ expr')
3  end
4  Function eval_assign_arbitrary_data_forkless(Φ, x, y) is
5  |  addr := GetAddress()
6  |  Create in Φ a symbolic value named b_addr of size 1
7  |  Create in Φ a symbolic value named non_det_addr of size sizeof(y)
8  |  Φ := Φ ∧ (NUMBER_FAULTS ≜ NUMBER_FAULTS + b_addr)
9  |  new_constraint := (non_det_addr ≠ y) || ¬b_addr
10 |  Φ := Φ ∧ (ACTIVATION_CONSTRAINT ≜
   |   ACTIVATION_CONSTRAINT & new_constraint)
11 |  y' := ite b_addr ? non_det_addr : y
12 |  return Φ, y'
13 end
14 Function eval_assign_handler(Φ, x, y) is
15 |  if IsFaultLocationFilter() then
16 |  |  case ArbitraryDataFaultForkless do
17 |  |  |  Φ', y' := eval_assign_arbitrary_data_forkless(Φ, x, y)
18 |  |  |  return eval_assign_normal(Φ', x, y')
19 |  |  end
20 |  |  ...
21 |  else
22 |  |  return eval_assign_normal(Φ, x, y)
23 |  end
24 end
```

Arbitrary data faults. We implemented four different arithmetic encodings for arbitrary data faults, we show in table 6.13 those encodings and their corresponding activation constraints. We use b_{addr} to represent the activation variable and non_det_{addr} to represent the fault effect. Note that *ite*s are written with a C-like syntax $ite\ c\ ?\ t\ :\ e$ where c is the boolean condition that if true, the expression results in the expression t and if false, the *ite* expression results in the expression e; $\oplus$ is used to denote a bitwise xor operator. We omit the extension operations from boolean to variable size value for readability. Forkless arbitrary data fault with the *ite* operator is illustrated in Algorithm 5.1.

Reset Faults. Similar to arbitrary data faults, we implement a reset fault model. The differences lie in the fault effect, which does not require a symbolic variable to represent it but is simply the value 0 or an expression resulting in 0 if b_{addr} is active, and the activation constraint that is adapted to exclude fault location assigning the value 0. We also implemented four different forkless encodings with various operators. There

Table 5.3: Arbitrary data encodings and their associated activation constraint

Fault model	Fault expression	Activation constraint
None	$x := y$	
Inlined if-then-else	$x := ite\ b_{addr}\ ?\ non\_det_{addr}\ :\ y$	$(non\_det_{addr} \neq y)\ ||\ \neg b_{addr}$
Multiplication	$x := y + b_{addr}\ \times non\_det_{addr}$	$(non\_det_{addr} \neq 0)\ ||\ \neg b_{addr}$
Bitwise and	$x := y + (-b_{addr})\ \&\ non\_det_{addr}$	$(non\_det_{addr} \neq 0)\ ||\ \neg b_{addr}$
Bitwise xor	$x := y \oplus (-b_{addr})\ \&\ non\_det_{addr}$	$(non\_det_{addr} \neq 0)\ ||\ \neg b_{addr}$

are details in Table 5.4.

Table 5.4: Reset encodings and their associated activation constraint

Fault model	Fault expression	Activation constraint
None	$x := y$	
Inlined if-then-else	$x := ite\ b_{addr}\ ?\ 0x0000\ :\ y$	$(y \neq 0x0000)\ ||\ \neg b_{addr}$
Subtraction	$x := y - b_{addr}\ \times y$	$(y \neq 0x0000)\ ||\ \neg b_{addr}$
Bitwise and	$x := y - (-b_{addr})\ \&\ y$	$(y \neq 0x0000)\ ||\ \neg b_{addr}$
Bitwise xor	$x := y \oplus (-b_{addr})\ \&\ y$	$(y \neq 0x0000)\ ||\ \neg b_{addr}$

Here, the activation constraint is always the same: the original right-hand side expression should not be 0.

Bit-flip Faults. The implementation of the bit-flip fault model is again very similar to the arbitrary data fault process. Two symbolic variables are created for the activation and the effect, their combination is different to result in a bit-flip, as shown in Table 5.5.

Table 5.5: Bit-flip encoding and its activation constraint

Fault model	Fault expression	Activation constraint
None	$x := y$	
Inlined if-then-else	$x := ite\ b_{addr}\ ?$ $y \oplus (1 << non\_det_{addr})\ :\ y$	$(non\_det_{addr} < sizeof(y))$ $\|\|\ \neg b_{addr}$

We only implemented the *ite* encoding for bit-flip but others can be imagined. The activation constraint limits the value of non_det_{addr} with respect to the size of the right-hand side expression to avoid overflows. This encoding of bit-flips allows only one bit-flip per assignment, even in a multi-fault context. This represents real classes of attacks, such as the Rowhammer attack. We believe an extended bit-flip encoding can be devised using masks to encode the possibility of multiple bit-flips in one fault location.

5.3.3.2 Test Inversion Faults

The implementation of the test inversion fault model differs from the implementation of other data faults. The assignment handler takes the normal –non-faulted assignment function. The faulty behavior is instantiated in the conditional jump handler. When the test inversion behavior is selected, it computes the new, faulted condition to give to the normal conditional jump function. This is illustrated in Algorithm 5.2.

Algorithm 5.2: Implementation sketch of a forkless test inversion process

Input: path predicate Φ, conditional jump instruction $if\ cdt\ l_t\ else\ l_e$
Data: worklist WL, fault counter $NUMBER\_FAULT$ and fault budget MAX_FAULT stored in Φ
Output: Updated Φ and WL

```
1  Function eval_assign_normal(Φ, cdt, l_t, l_e) is
2  |  if Φ ∧ cdt ∧ (NUMBER_FAULTS ≤ MAX_FAULT) is satisfiable then
3  |  |  Add (Φ ∧ cdt, l_t) to WL
4  |  end
   |  /* Idem for else branch (¬cdt)                                  */
5  end
6  Function eval_test_inversion_forkless(Φ, cdt) is
7  |  addr := GetAddress()
8  |  Create in Φ a symbolic value named b_addr of size 1
9  |  Φ := Φ ∧ (NUMBER_FAULTS ≜ NUMBER_FAULTS + b_addr)
10 |  cdt' := ite b_addr ? ¬cdt : cdt
11 |  return Φ, cdt'
12 end
13 Function eval_condition_jump_handler(Φ, cdt, l_t, l_e) is
14 |  if IsFaultLocationFilter() then
15 |  |  case TestInversionForkless do
16 |  |  |  Φ', cdt' := eval_test_inversion_forkless(Φ, cdt)
17 |  |  |  return eval_conditional_jump_normal(Φ', cdt', l_t, l_e)
18 |  |  end
19 |  |  ...
20 |  else
21 |  |  return eval_conditional_jump_normal(Φ, cdt, l_t, l_e)
22 |  end
23 end
```

There is no need for an activation constraint in this fault model, as the bitwise negation of a boolean value is always its opposite. We used the *ite* operator but other forkless encodings are possible.

5.3.3.3 Instruction Skip Faults

The last fault model we implemented is the instruction skip fault model. The idea is to be able to create a nop (no-operation) with a forkless encoding. As each kind of DBA instruction can be the target of a skip, the remaining ones (for static jumps and dynamic jumps) are refactored with a wrapper and a fault filter like the conditional jump handler. A summary of the fault effect for each DBA instruction is presented in Table 5.6, where *addr* represents a distant code address, *next* is the next instruction in the memory layout and *expr* corresponds to an expression computing one or multiple addresses. No activation constraint is required. Except for assignments, the other faulted instructions are likely to open new paths.

Faulting a DBA Block. The instruction skip fault model applies to an entire DBA

Table 5.6: Instruction skip encoding for each dba instruction

DBA instruction	Fault model	Fault expression
Assignment	None	$x := y$
	Forkless Skip	$x := ite\ b_{addr}\ ?\ x\ :\ y$
Conditional jump	None	if cdt goto $addr$ else goto $next$
	Forkless Skip	if $(cdt\ \&\ \neg b_{addr})$ goto $addr$ else goto $next$
Static jump	None	goto $addr$
	Forkless Skip	if b_{addr} goto $next$ else goto $addr$
Dynamic jump	None	goto $expr$
	Forkless Skip	goto $(ite\ b_{addr}\ ?\ next\ :\ expr)$

block, not just one DBA instruction, to correspond to one ISA instruction being skipped. This is relevant only for assignment instructions that need more than one instruction for flags and register updates.

Assignment. The Forkless encoding of the instruction skip fault model acts for assignments as a data fault, with the effect corresponding to the old value of the right-end-side variable. This is performed for all variables involved in the computation, may it be flags, register or memory operations, with the same activation variable. The difficulty is to know if one instruction in the DBA block is blacklisted (if it updates a variable that should always remain without faults). If so, the fault needs to be canceled for all previous instructions in the block as they were optimistically injected with faults. In practice, the variable containing the activation for the whole bloc is assigned the value 0 in the path predicate. An assignment DBA block ends with a static jump to the next instruction, we reset the cancellation flag and the shared activation variable at this stage.

Conditional Jump. Here, the condition is augmented with the possibility of a fault before it is given to the normal conditional jump process.

Static Jump. This instruction is in effect replaced by a conditional jump, with the fault activation variable as the condition.

Dynamic Jump. The expression that results in the target address of the jump is faulted as a data expression would be.

5.3.4 Early Detection of Fault Saturation (EDS)

Our first optimization, EDS aims at stopping injection as soon as possible. The implementation follows the algorithm described in Section 4.4.1. Note that this optimization has been designed and implemented for data faults only. Test inversion contains very few fault locations in comparison. We believe it can straightforwardly be adapted for instruction skips.

A new behavior is added to the conditional jump handler, where the satisfiability query is first asked for $nb_f < max_f$ and then for $nb_f == max_f$ to detect the fault saturation. We created an internal flag value to register the activation of EDS and not inject fault when the flag is true.

We encountered a challenge implementing EDS. Imagine some faults F are injected, then a conditional jump where no faults is needed to take a certain branch A. Then the constraint $nb_f < max_f$ is recorded in the path predicate. Imagine the path encounters

another conditional jump, where one of its branches, B, needs a fault in F. In the context of a single-fault attacker model, taking the branch A constrains F to be inactive since the sum of the faults before A needs to be 0, hence preventing the exploration of branch B, even if it is possible for the attacker model.

To solve this issue, we modified the implementation of the function performing the solver call. An additional argument was added, containing the constraints that shouldn't be added to the path predicate but that are still part of the query. We call this parameter *such_that*: can the path predicate and the conditional condition be true *such that* this parameter is also true.

5.3.5 Injection On Demand (IOD)

Our second optimization, IOD, described in Section 4.4.2, aims at starting injection as late as possible, and only adding new faults when necessary. Note that again, this optimization has been designed with data faults. Test inversion contains very few fault locations in comparison. We show later how it was adapted for instruction skips, and focus the first part of the description on its application to data faults.

Contrary to EDS, IOD required heavy refactoring. The injection on demand requires to compute and keeping in parallel two path predicates. We name *dual* the combined symbolic states. All functions have to be updated with this new type. Either they need only one symbolic state and the caller function has to be modified to give the relevant symbolic state, or the function receives a dual and its logic must be adapted to take the dual into account.

The initialization instantiates an extra symbolic state. Path predicate update functions need to perform the same operations twice, once on the normal predicate and once on the faulted predicate. This includes assignments, conditional jumps, static jumps and dynamic jumps, but also all functions interacting with the SMT solver and the directive handlers. A new internal variable was created to keep track of the under-approximation counter. Its value is checked before injecting faults.

The assignment process is duplicated, the normal path predicate is updated with the normal assignment behavior, and the faulted path predicate is updated with the faulty behavior presented above.

IOD for Instruction Skip. All types of instructions need to be taken into account for this fault model.

- We keep the same mechanism for assignments and conditional jump, though adding the possibility of skipping the latter;
- We never merge for static jumps, as previous instruction skips do not influence the static jump address;
- We systematically merge for dynamic jumps. This is a choice different than for data faults where paths are not merged there. As we do not have an efficient algorithm for faults on addresses, this choice for data faults limits faults on dynamically computed addresses. However, the forkless instruction skip doesn't have limitations on skipping address assignments or on faults regarding addresses in general, all instructions can be skipped.

5.3.6 Sub-fault Simplification

To improve performance further by reducing the number of injection points, we devised a new optimization, sub-fault simplification. The idea is to detect faults subsumed by

another fault, especially in the case of arbitrary data faults. The dominated fault can be nullified without loss of generality.

```
x := 8
x := x + 1
```

(a) Original statement

```
x:= ite fault_here_1 ? fault_value_1 : 8
x:= ite fault_here_2 ? fault_value_2 :
            (fault_here_1 ? fault_value_1 : 8) + 1
```

(b) Forkless transformation for arbitrary data fault

Figure 5.3: Illustration of sub-fault simplification

An example of this pattern is provided in Figure 5.3. The same left-hand-side variable is assigned multiple times, only the last one could be faulted with an arbitrary data fault to reduce fault injection points.

We implemented this concept as a query preprocessing, adding the expression nullifying fault activation variables in queries. We designed two variants:

- one where the dominated faults are detected on a per-query basis ($Free$),
- and another using a memoization technique in the hope of improving performance (Mem).

However, a dominated fault in one query is not necessarily dominated in another, hence there are some soundness issues with this technique. We detect unsound cases and propose two recovery mechanisms:

- *Abort* where the SFS simplification is not added to the query,
- and *Lazy* still trying the query and sending it again without SFS if the query is UNSAT.

Unfortunately, experiments showed (see Section 6.3.2) that this optimization was costly in execution time due to query traversal. It was then abandoned.

5.3.7 Forking Fault Models

In order to compare our novel forkless encoding technique against the state-of-the-art techniques in the fairest way, we implemented forking encodings alongside FASE. Note that mutant generation scales worst. This allows us to restrict the comparison to the encodings only, the rest of the analysis workflow being the same inside BINSEC.

Forking faults appear as other faulting options inside the instruction handler. Previously described optimizations do not apply to forking faults as their placements are concrete.

5.3.7.1 Forking Data Faults

This behavior differs from forkless data faults as the path is split between a path with faults for this instruction and one without. This is similar to a normal conditional jump. Both paths are stored in BINSEC's worklist WL and one is dequeued to continue the exploration inside the wrapper. Algorithm 5.3 illustrates the general process for arbitrary data fault.

The forking encodings split the path in two, in one path the activation variable b_{addr} is true, the assignment is faulted and the fault counter is incremented; in the second path, the assignment proceeds with the normal behavior. We perform the number of faults check at each fault location for forking faults because the number of faults is concrete and it allows pruning unfeasible branches early on.

The arbitrary data fault model simply replaces the right-hand side of the assignment with a nondeterministic value. The reset fault model is identical except it replaces the right-hand side with zeros, and bit-flips flip a bit with an encoding similar to the forkless one.

5.3.7.2 Forking Test Inversion Faults

The implementation of the forking test inversion fault model follows the same principles as the forking data fault models, except it takes place in the conditional jump handler. A verification of the number of faults already performed is done, then the path is split in two, one will execute the normal conditional jump and the other one, a conditional jump with the negation of the condition.

5.3.7.3 Forking Instruction Skip Faults

The instruction skips fault model also has its forking counterpart implemented, still following the same principles. Here, when the fault on an assignment block has to be canceled because it updates a black-listed variable, the exploration is simply stopped in the faulted branch and continues in the branch without the fault.

5.3.8 Conclusion

We saw in this section how BINSEC was used to build an Adversarial Symbolic Execution tool, BINSEC/ASE, able to assess the adversarial reachability of a location in a binary program. Most of BINSEC is kept as is, the main changes happen at the path predicate computation.

We will now take a step back from technical details and explore a methodology to analyze a new program as a user guide and a methodology to add a new fault model as a developer guide.

Algorithm 5.3: Implementation sketch of a forking arbitrary data fault assignment process

Input: path predicate Φ, assignment instruction $x := y$
Data: fault counter $NUMBER\_FAULT$ and fault budget MAX_FAULT stored in Φ
Output: Updated Φ

```
1  Function eval_assign_normal(Φ, x, y) is
2  |  return Φ ∧ (x ≜ expr')
3  end
4  Function eval_assign_arbitrary_data_forking(Φ, x, y) is
5  |  if Φ ∧ (NUMBER_FAULTS ≤ MAX_FAULT) is satisfiable then
6  |  |  addr := GetAddress()
7  |  |  Create in Φ a symbolic value named b_addr of size 1
8  |  |  Create in Φ a symbolic value named non_det_addr of size sizeof(y)
9  |  |  Split in Φ the values for variable b_addr
10 |  |  case 0b1 do
11 |  |  |  Φ' := Φ ∧ (b_addr ≜ 0b1)
12 |  |  |  Φ' := Φ' ∧ (NUMBER_FAULTS ≜ NUMBER_FAULTS + 1)
13 |  |  |  new_constraint := (non_det_addr ≠ y) || ¬b_addr
14 |  |  |  Φ' := Φ' ∧ (ACTIVATION_CONSTRAINT ≜
   |  |  |    ACTIVATION_CONSTRAINT + new_constraint)
15 |  |  |  y' := non_det_addr
16 |  |  |  return eval_assign_normal(Φ', x, y')
17 |  |  end
18 |  |  case 0b0 do
19 |  |  |  Φ' := Φ ∧ (b_addr ≜ 0b0)
20 |  |  |  return eval_assign_normal(Φ', x, y)
21 |  |  end
22 |  else
23 |  |  eval_assign_normal(Φ, x, y)
24 |  end
25 |  return Φ', y'
26 end
```

5.4 User Guide: a Methodology to Analyse a New Program

The aim of this section is to provide a brief guide to the interested user so they can use BINSEC/ASE on their own programs, from the setup to the evaluation phase. In particular, this section does not aim to be exhaustive but to give an idea of the engineering effort required to perform an analysis and list the main steps. An overview is resented in Figure 5.4.

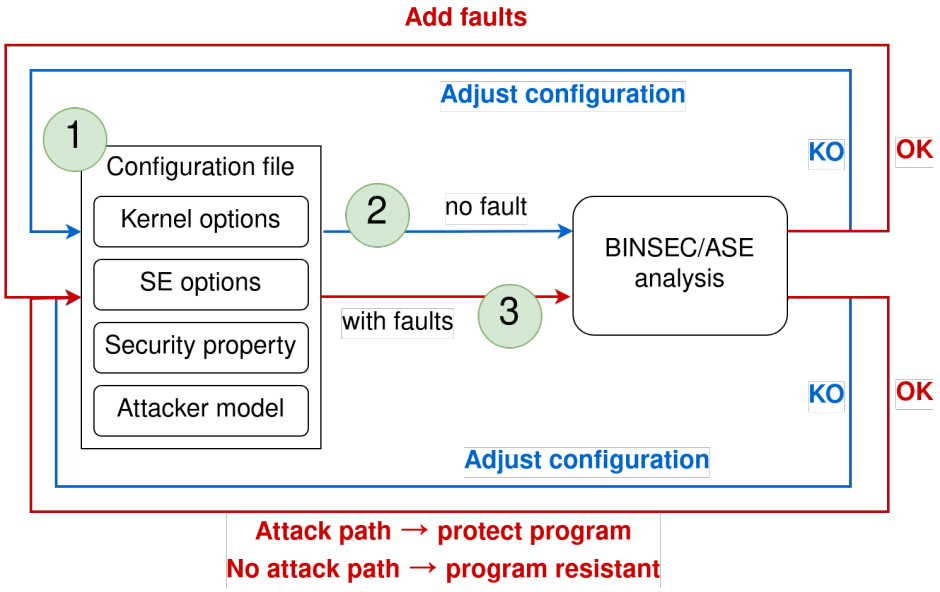


Figure 5.4: User guide workflow

5.4.1 Running Example

To illustrate the usage of BINSEC/ASE, let's consider again the basic version of the VerifyPIN program that we saw in Chapter 2. It is presented in Figure 5.5, with some PIN initialization (lines 7 to 17). We assume the attacker does not know the correct PIN, so they try one at random, "0 0 0 0" for instance. The attacker goal is to be authenticated, i.e. to pass the assertion line 41.

The program works as follows. The *main* function line 38 starts by initializing the PINs, and then it calls the $verifyPIN$ function. This function resets the $g\_authenticated$ variable to false by default, then calls the $byteArrayCompare$ function to compute the PIN check. If $byteArrayCompare$ returns 1, then the program authenticates the user. Inside $byteArrayCompare$, each digit is checked against the card PIN digit and if one is detected to be different, the function returns 0.

Note that this program is very naive and, in particular, not constant time. Other versions are proposed in FISSC [DPP+16] that are. Here, we only consider an attacker able to inject faults into a program, and not perform side-channel analysis.

We compile toward Intel x86-32 bits. As we use a previous version of BINSEC (2021), support for Intel x86-64 bits is not available. It becomes available in newer BINSEC versions. Note that our technique is architecture independent and considering 32-bit programs still makes sense in an embedded system context.

This C program is compiled with *gcc* using the following command line:

```
1 #define PIN_SIZE 4
2
3 bool g_authenticated = 0;
4 int g_userPin[PIN_SIZE];
5 int g_cardPin[PIN_SIZE];
6
7 void initialize() {
8     int i;
9     // card PIN = 1 2 3 4
10    for (i = 0; i < PIN_SIZE; ++i) {
11        g_cardPin[i] = i+1;
12    }
13    // user PIN = 0 0 0 0
14    for (i = 0; i < PIN_SIZE; ++i) {
15        g_userPin[i] = 0;
16    }
17 }
18
19 bool byteArrayCompare(int* a1, int* a2, int size) {
20     int i;
21     for(i = 0; i < size; i++) {
22         if(a1[i] != a2[i]) {
23             return 0;
24         }
25     }
26     return 1;
27 }
28
29 void verifyPIN() {
30     g_authenticated = 0;
31     if(byteArrayCompare(g_userPin, g_cardPin,
32         PIN_SIZE) == 1) {
33         g_authenticated = 1; // Authentication();
34     }
35     return;
36 }
37
38 void main() {
39     initialize();
40     verifyPIN();
41     assert(g_authenticated == 1);
42 }
```

Figure 5.5: Running example, inspired by VerifyPIN [DPP+16]

```
$ gcc -m32 -static -O0 -g verifyPIN_0.c
```

The flag *m32* indicates to compile into a 32 bits binary, *static* means all dependencies are statically linked which is required by the BINSEC tool, the optimization level *O0* and the inclusion of debug symbols with the $-g$ flag allow for easy mapping between source code and binary program.

5.4.2 Analysis Goal

An analysis with BINSEC/ASE starts by defining what type of security property the user wants to study and what type of attacker the program faces.

Depending on the analysis goal, the exploration can be configured through a dedicated option to stop at the first attack found if any, or to explore exhaustively all adversarial paths in the program. If the program is resistant to the considered attacker model, the analysis being k-complete, it explores all paths up to a bounded depth to ensure no bounded attack paths are missed. It is sometimes possible to increase the depth bound to get the completeness of the analysis.

5.4.2.1 Security Properties

Security properties are application-specific and finding relevant ones is an active area of research. It is not in the scope of this thesis. However, we provide the following few pointers.

Many different properties can be important to ensure in a program like authentication, data integrity or correct program output in general. When verifying an authentication program, an interesting property is to make sure the password given is indeed the reference password. For cryptographic programs, there may be mathematical formulas that should remain true to avoid leaking the secret key like the BelCore attack in CRT-RSA. The attacker's goal is to violate that security property.

BINSEC/ASE supports properties that can be expressed with existing program variables, or even ghost variables that the user updates through stubs. The security oracle can also be written in DBA as part of the configuration file, then it can be expressed with all DBA variables involved, and in particular it grants access to flags, resisters and memory locations. We can even imagine verifying trace properties. In the examples we provide, the property is expressed with program variables as an 'assert' in the program.

Once the desired security property is established, there remains the question of the location in the program where it should be true. It may be at the end of the program to verify a program's output or in the middle, checking for memory access out of bounds for instance. To increase the efficiency of the analysis, we suggest identifying locations in the code from where the security property cannot be violated anymore to prune the exploration. It can correspond to some program returns or calls to library functions handling error cases.

Running example. In the VerifyPIN program, we consider that the attacker goal is to be authenticated despite not having the correct PIN. At the program level, it corresponds to reaching the true branch of the assertion in the binary (address 0x08049e3a) corresponding to line 41 in C. We place a *cut* at the false branch of the assertion, at address 0x08049e1e, since the attacker cannot succeed past that point. Figure 5.6 shows the assembly code for the *main* function.

```
08049deb <main>:
8049deb:        endbr32
8049def:        lea    0x4(%esp),%ecx
8049df3:        and    $0xfffffff0,%esp
8049df6:        pushl  -0x4(%ecx)
8049df9:        push   %ebp
8049dfa:        mov    %esp,%ebp
8049dfc:        push   %ebx
8049dfd:        push   %ecx
8049dfe:        call   8049bc0 <__x86.get_pc_thunk.bx>
8049e03:        add    $0x9b1fd,%ebx
8049e09:        call   8049ce5 <initialize>
8049e0e:        call   8049da7 <verifyPIN>
8049e13:        movzbl 0x12dc(%ebx),%eax
8049e1a:        cmp    $0x1,%al
8049e1c:        je     8049e3a <main+0x4f>
8049e1e:        lea    -0x30fd4(%ebx),%eax
8049e24:        push   %eax
8049e25:        push   $0x2b
8049e27:        lea    -0x30ff8(%ebx),%eax
8049e2d:        push   %eax
8049e2e:        lea    -0x30fea(%ebx),%eax
8049e34:        push   %eax
8049e35:        call   804aff0 <__assert_fail>
8049e3a:        nop
8049e3b:        lea    -0x8(%ebp),%esp
8049e3e:        pop    %ecx
8049e3f:        pop    %ebx
8049e40:        pop    %ebp
8049e41:        lea    -0x4(%ecx),%esp
8049e44:        ret
```

Figure 5.6: Running example: disassembly of the main function

5.4.2.2 Attacker Model

Once the security property to verify has been defined, the relevant attacker model has to be specified. Note that a very powerful attacker model can break almost any security property, hence the need to choose the right strength level, after a risk analysis for the application.

Attacker Capability and Attacker Budget. Those two parameters define the strength of the attacker model and can give an idea of the injection means at their disposal. They should be selected according to the risk analysis previously performed.

Target Locations. This corresponds to the set of instruction addresses that can be faulted by the attacker. It is expressed as a list of intervals. For large programs, we recommend focusing the target locations on the critical sections of code to get faster results. In a single fault scenario, the analysis can be split with a different section targeted each time. If the security oracle has been added to the code, we recommend

excluding that part from the target locations.

This can also help refine the attacker model. If an attacker model can only fault some specific instructions, target locations can be enumerated exhaustively. For instance, skipping only function calls can be modeled by listing as targets only the function call with an instruction skip fault model. Faulting data only on loads and stores, at the memory interface, can be represented in this way too. In those two examples, it is possible to design specific fault models, we refer the interested reader to Section 5.5.

Other Fault Modifiers. To prevent faults on variables deemed secure, not interesting, or with no efficient algorithm to handle them, a target blacklist can be used. We mainly use it for the *esp* and *gs* registers in Intel x86 binaries. The *esp* register is the stack pointer and we do not currently have an efficient algorithm to handle faulting it. The *gs* resister is used as a base pointer to point to a data region in some programs. Faulting *gs* results in a combinatorial explosion as the base pointer for data can now be taken anywhere in a data fault model. In general, we do not have an efficient algorithm to handle faults on memory addresses, those can be filtered out with a configuration option.

By default, BINSEC/ASE doesn't fault flags as it would largely increase the number of fault locations and new symbolic variables. Faults on 'normal' variables are most of the time expressive enough with respect to the attacker model. However, it is possible to activate faults on flags with a configuration option.

Running example. Let's consider an attacker model able to perform arbitrary data fault for instance. Since the VerifyPIN program does not have any protection, we expect it to be vulnerable to an attacker able to inject one fault. For target locations, we exclude the *initialize* and *main* functions. This leaves the $verifyPIN$ and *byteArrayCompare* functions (assembly code in Figures 5.7 and 5.8), where we also exclude the beginning which put the return address onto the stack, as well as the end of the functions handling the return process. It leaves two ranges of addresses to fault: 0x08049dba to 0x08049de6 in $verifyPIN$ and 0x08049d5e to 0x08049da0 in *byteArrayCompare*.

```
08049da7 <verifyPIN>:
8049da7:        endbr32
8049dab:        push   %ebp
8049dac:        mov    %esp,%ebp
8049dae:        push   %ebx
8049daf:        call   8049bc0 <__x86.get_pc_thunk.bx>
8049db4:        add    $0x9b24c,%ebx
8049dba:        movb   $0x0,0x12dc(%ebx)
8049dc1:        push   $0x4
8049dc3:        mov    $0x80e6c94,%eax
8049dc9:        push   %eax
8049dca:        mov    $0x80e6c84,%eax
8049dd0:        push   %eax
8049dd1:        call   8049d4a <byteArrayCompare>
8049dd6:        add    $0xc,%esp
8049dd9:        cmp    $0x1,%al
8049ddb:        jne    8049de5 <verifyPIN+0x3e>
8049ddd:        movb   $0x1,0x12dc(%ebx)
8049de4:        nop
8049de5:        nop
8049de6:        mov    -0x4(%ebp),%ebx
8049de9:        leave
8049dea:        ret
```

Figure 5.7: Running example: disassembly of the verifyPIN function

```
08049d4a <byteArrayCompare>:
8049d4a:        endbr32
8049d4e:        push   %ebp
8049d4f:        mov    %esp,%ebp
8049d51:        sub    $0x10,%esp
8049d54:        call   8049e45 <__x86.get_pc_thunk.ax>
8049d59:        add    $0x9b2a7,%eax
8049d5e:        movl   $0x0,-0x4(%ebp)
8049d65:        jmp    8049d98 <byteArrayCompare+0x4e>
8049d67:        mov    -0x4(%ebp),%eax
8049d6a:        lea    0x0(,%eax,4),%edx
8049d71:        mov    0x8(%ebp),%eax
8049d74:        add    %edx,%eax
8049d76:        mov    (%eax),%edx
8049d78:        mov    -0x4(%ebp),%eax
8049d7b:        lea    0x0(,%eax,4),%ecx
8049d82:        mov    0xc(%ebp),%eax
8049d85:        add    %ecx,%eax
8049d87:        mov    (%eax),%eax
8049d89:        cmp    %eax,%edx
8049d8b:        je     8049d94 <byteArrayCompare+0x4a>
8049d8d:        mov    $0x0,%eax
8049d92:        jmp    8049da5 <byteArrayCompare+0x5b>
8049d94:        addl   $0x1,-0x4(%ebp)
8049d98:        mov    -0x4(%ebp),%eax
8049d9b:        cmp    0x10(%ebp),%eax
8049d9e:        jl     8049d67 <byteArrayCompare+0x1d>
8049da0:        mov    $0x1,%eax
8049da5:        leave
8049da6:        ret
```

Figure 5.8: Running example: disassembly of the byteArrayCompare function

5.4.3 Configuration

For detailed configuration options, we refer the interested reader to the online documentation and examples provided as part of the BINSEC/ASE artifact[9].

Running Example. Figure 5.9 shows what a configuration file for the VerifyPIN program would look like, with the described attacker model. The memory file *mem* only contains a starting address for the stack pointer. An appropriate value for the maximal exploration depth *depth* is discussed in Section 5.4.5.

```
esp<32> := 0xFFFFFF00;
```

```
[kernel]
isa = x86
entrypoint = main
file = verifyPIN_0.x

[sse]
enabled = true
depth = 200
memory = mem

fault-model = ArbitraryData
max-faults = 1
goal-address = 0x08049e3a
assert-fail-address = 0x08049e1e
target-addresses = 0x08049dba,0x08049de6,0x08049d5e,0x08049da0
injection-method = on-demand
target-blacklist = esp
where-check-nb-faults = branch
```

Figure 5.9: Running example: BINSEC/ASE configuration file

5.4.4 Reading BINSEC/ASE Output

We base this section on the output of BINSEC/ASE for the VerifyPIN running example.

Run Statistic. The output ends with some analysis statistics (see Figure 5.10). For instance, We can see on line 2 that the analysis explored 5 different paths, analyzed 206 instructions in total (line 5) and the longest path is 163 instructions deep (line 6).

To determine if the analysis was complete (as opposed to k-complete), the following statistics should be checked:

- The number of 'cut early paths' line 8 should be 0;
- The number of SMT queries the solver answered 'unknown' for should be 0 (line 20).

The 'SMT queries' section provides a number of information about queries and reliance on the SMT solver. It gives an idea about the query complexity handled by the analysis.

[9] https://github.com/binsec/esop2023_artefact

- Line 11, we see that 61 queries were generated but only 13 of them needed to be sent to the solver (line 12). The difference has been arithmetically simplified into true or false (lines 18 and 19 for assume queries);
- The 'Average solving time' (line 13) gives in seconds the average time the solver took to answer per query. Here, our tool wasn't precise enough to record the small value, hence the display of a 0.

Among the 'Fault injection' statistics, we have a reminder of the selected attacker model options (lines 23, 25, 26 and 27) and chosen optimizations (lines 24 and 28). The total number of injections performed is indicated in line 29. In the end, resides overall analysis results:

- 'Models found' (line 33) indicates the number of adversarial paths produced by BINSEC/ASE;
- 'Assert cut' (line 34) relates to the number of paths where the attacker goal was not adversarially reachable.

Attack Path. We explore in Figure 5.11 one of the produced attack paths to understand how to read that part of the BINSEC/ASE output. Line 1 indicates that the goal address 0x08049e3a has been reached and the solver proposed the model following. Only 3 different faults were injected in that path and the fault at the address 0x08049d5e is the activated one (line 4). It corresponds to the initialization of the loop counter i, equivalent to line 20 in Figure 5.5. The solver proposed to inject the value 0x7ffffffc (line 9) instead of the original value 0. Since 0x7ffffffc is strictly greater than PIN_SIZE, the program does not enter the loop and returns 1 as if all digits were correct. Hence, injecting this fault allows the attacker to reach its goal.

```
1 [sse:info] Exploration
2         Paths 5
3         Paths continuing after max reached 0
4         Secondary queries (EDS only) 0
5         Analysed instructions 206
6         Max depth instruction 163
7         Max depth branchments 16
8         Cut early paths 0
9
10 [sse:info] SMT queries
11        Number of queries 61
12        Number of queries sent to the solver 13
13        Average solving time 0.000000
14        Total assume queries 49
15        Assume queries sent to solveur 13
16        Total enumerate queries 12
17        Enumerate queries sent to solveur 0
18        Trivial true assume 18
19        Trivial false assume 18
20        Unknown solver 0
21
22 [sse:info] Fault injection
23        Fault model ArbitraryData
24        Injection method on-demand
25        Maximum number of faults 1
26        Where number of faults checked: branch
27        Where non_det constrains checked: branch
28        At branch optim: false
29        Injection locations 15
30        Min Injection locations per path 0
31        Max Injection locations per path 15
32        Avg Injection locations per path 11
33        Models found 3
34        Assert cut 2
```

Figure 5.10: Running example: BINSEC/ASE statistics output

```
1 [sse:result] Model 08049e3a
2     --- Model ---
3     # Variables
4     b_8049d5e : #b1
5     b_8049d98 : #b0
6     b_8049dba : #b0
7     ebp : --
8     ebx : --
9     non_det_8049d5e : #x7ffffffc
10    non_det_8049d98 : #x7ffffffd
11    non_det_8049dba : #x00
12
13    -- empty memory --
```

Figure 5.11: Running example: BINSEC/ASE attack path

5.4.5 Analysis Process

We describe here the recommended process when adding a new benchmark, after the security property and attacker model definition step has been performed. This is an iterative process to ensure correct configuration and early results.

As configuring a BINSEC/ASE analysis may be a trial-and-error process, we recommend starting with a non-faulted run. This helps to see if there is any warning or error to resolve. If an attack is found, it should be investigated. Either there is an existing bug in the program or the configuration is not adapted. Other statistics can be read and examined, like the number of explored paths or solver statistics. As a convenience, we use the maximal depth of a non-faulted run, rounded up to the hundred as a bound for faulted runs, this metric can be recovered at this step. The default maximal depth setting may need to be increased to adapt to the program.

The next step is to add the attacker model to the configuration. The analysis is likely to take longer and explore more paths. We recommend starting from a rather weak attacker model, able to perform only one fault, then analyzing the BINSEC/ASE run, before incrementally strengthening the attacker model until the desired one. This incremental process enables the detection of any abnormal behavior, warnings or errors before the analysis gets too complex. If an attack is found for a weaker attacker model than the desired one, the program is vulnerable. The attack can be analyzed and possibly protected against.

5.4.6 Summary

We presented in this section a methodology to set up a new program for a BINSEC/ASE analysis. Here is a summary of the different steps.

1. Express the security properties of the program;
2. Decide on an attacker model relevant to the program;
3. Write the configuration file;
4. Run BINSEC/ASE without faults to verify the configuration;
5. Incrementally increase the strength of the attacker model until the desired one.

Now that the program analysis is set up as intended, the results can be integrated

into the bigger picture, for instance, a security evaluation by a security practitioner.
- If the program was vulnerable when it was not supposed to be, a strengthening process can begin, adding countermeasures and starting the analysis process presented above, iterating until the analysis results meet the security requirements;
- If the program proves resistant to the considered attacker model, it can be interesting to consider other properties the program should have, and start the analysis process again with this new attacker model.

5.5 Developer Guide: a Methodology to Add a New Fault Model

The objective of this section is to present a methodology to add a new fault model inside BINSEC/ASE. We acknowledge that there exist many different fault models. We implemented some in this thesis as proof of concepts, but others may be more relevant to a specific application. This guide is meant to provide a starting point for interested developers and does not aim to be exhaustive.

Running Example. In this section, the instruction skip fault model is the running example. It was the last to be implemented and required modifications in many parts of the symbolic engine.

5.5.1 Defining the New Fault Model

The first step to adding a new fault model is to design it on paper. We discuss it in this section.

5.5.1.1 State-of-the-art Attack

A fault model is usually derived from a capability given to an attacker by a real attack that a security practitioner wishes to simulate. To represent this attack, a developer needs to understand it in terms of:
- what type of capability it gives to an attacker (what action on what object);
- when it can be applied in the execution;
- in which part of the system the fault is introduced;
- does multiple faults make sense and if so how.

Running Example. Instruction skip is a common model used to describe the effects of hardware fault injection on programs. In practice, instructions can be modified so they don't have side effects, or the prefetch buffer may be tempered with. A skip can occur on any ISA instruction. Multiple separated faults are possible in the state-of-the-art, as well as skipping multiple consecutive instructions.

5.5.1.2 Fault Model

The details of the attacker capability to model have to be expressed in terms of their impact on the path predicate computation. That is to say, how they modify the path predicate initialization, the adversarial reachability query and the behavior of the four main DBA instructions: assigns, static, conditional and dynamic jumps. At this step, the capability is often over-approximated for simplicity. It is possible to imagine complex fault models with interactions between different instructions. It can be the

case when considering precisely the micro-architecture [TAC+22]. It would require keeping track of a trigger flag in the symbolic state.

Running Example. We modeled instruction skip as not executing an instruction entirely, not updating the memory, the registers or the flags. Forkless instruction skip has the following impact on the path predicate computation (see Section 5.3.3.3 for more details).

- *Assign*: a data fault is injected, with the corrupted value being the old value of the variable.
- *Static jump*: this instruction is transformed into a conditional jump, with the fault activation as the condition, the next instruction directly below in addresses as the target for the true branch, and the original target for the else branch.
- *Conditional jump*: the condition is augmented with an activation variable, that if true, the analysis goes to the 'then' branch which corresponds to the next address in the memory layout.
- *Dynamic jump*: the expression providing the possible jump targets is augmented similarly to a data fault, adding the next instruction address as a jump target possibility.

Here, we see an example of a fault model modifying all the DBA instructions, it is not necessarily the case. For instance, a new kind of data fault would only modify the assignment process.

We choose to implement multiple distinct faults to represent an attacker with a multiple instruction skip capability, with the existing machinery (number of faults check in particular). We believe implementing a fixed number of consecutive faults can be done by storing fault activation variables and a timer in the path predicate, adding with a logical *and* the stored activation variables to the current instruction's activation and decreasing the timer at each time until their timer reaches 0.

5.5.2 Implementation

Once the plan for which instruction handler needs to be modified and how, it can be implemented inside BINSEC/ASE. A new fault model may need other adjustments, such as:

- the initialization process,
- the directives handlers,
- the fault location filtering process already implemented may work or may need to be modified,
- it may also require additional internal variables stored in the path predicate, like counters, we have some for injected faults, or boolean flags, indicating a fault saturation for instance,
- queries sent to the SMT solver may be impacted in their form (adding new elements) and in which part of the query should be stored as a new constraint in the path predicate or not,
- the new fault model has to be added to the configuration option to allow a user to select it.

We provide Ocaml interfaces in BINSEC/ASE for the elements listed above in Table 5.7.

For comparison, we implemented a forking version of all our forkless fault models. If the developer wishes to do the same, a new fault model needs to be designed, as the impact of the forking fault model on the path predicate is different.

Table 5.7: Ocaml interfaces

Elements to modify	BINSEC/ASE Ocaml interface
Initialization process	src/ee/sse.ml:do_sse
Directive handlers	src/ee/sse.ml:[handle_assumptions, handle_reach], src/ee/sse.ml:[handle_enumerate, handle_cut]
Fault filter	src/ee/sse.ml:is_fault_location
Path predicate	src/ee/sse_types.[ml, mli]:Path_state
SMT queries	src/ee/senv.[ml, mli]:[assume, enumerate]
Options	src/ee/sse_options.[ml, mli]
Exploration metrics	src/ee/sse.ml:Stats
Solver metrics	src/ee/senv.ml:Query_stats

5.5.3 Dedicated Metrics

We recommend implementing dedicated metrics to monitor the new fault model. It can be the number of times it is triggered, a notion of where it is triggered, etc. This helps the debugging process and provides more details in the BINSEC/ASE output for further program or attack study. We distinguish two types of metrics, exploration metrics and solver metrics. Interfaces for them are listed in Table 5.7.

Running Example. For instance, the instruction skip fault model comes with counters for how many of each kind of DBA instruction has been faulted.

5.5.4 Testing

Implementing a new fault model is not trivial and requires testing to ensure the correctness of the implementation. We recommend writing a few minimal testing programs, each showcasing a different aspect of the fault model, for one and two faults. Then, configure them, run the BINSEC/ASE analysis and analyze manually the results. This means examining the number of paths explored and ensuring it is correct, seeing where queries have been created and that it is reasonable, verifying the total number of injection points, etc. And study the dedicated metrics added to be sure to understand what happens. BINSEC comes with different levels of debug logs which can be very helpful in understanding the unrolling of the analysis. If the implementation of the new fault model was somewhat 'invasive' and could potentially disturb other fault models, we recommend performing non-regression tests.

5.5.5 Summary

We presented in this section a methodology to add a new fault model. Here is a summary of the different steps.

1. Define what the fault model should represent (a type of attack for instance);
2. Define the fault model in terms of how it transforms DBA instructions;
3. Implement those transformations and other alterations to the symbolic engine;
4. Implement dedicated metrics;
5. Create testing programs and evaluate them to ensure the correctness of the implementation.

When all of those steps are done, the new fault model is ready to be used in a real security scenario.

5.6 Discussion

We discuss in this section considerations on what can or cannot be represented with BINSEC/ASE, in particular limits to fault model support.

5.6.1 BINSEC/ASE Limitations

We discuss here the main limitations of the BINSEC/ASE prototype.

5.6.1.1 Building on Top of an Existing Tool

The prototype presented in this chapter is based on BINSEC version 0.4.0, forked September 1^{st} 2021. BINSEC underwent a major refactoring shortly after, which made rebasing impossible, only a full re-implementation could have worked. This was not done for time reasons. Hence BINSEC/ASE does not benefit from recent developments in configuration syntax readability, architecture support (Intel x86-64 bits for instance) and other continued improvements. We leave as future work re-implementing the FASE engine in the current BINSEC version.

5.6.1.2 Supported Fault Models

Faults on Addresses. We do not have an efficient algorithm for faults on addresses. As many values are possible, the solver can lose time considering them and the possible effects of their propagation in the following instructions. Another issue is that the solver can select an address in an uninitialized part of the memory and fill it with exactly the convenient content, which is not very realistic. Hence, we chose to avoid faulting values 'looking like' address, with a configurable threshold. In practice, it also limits our ability to study faults related to stack manipulations.

Faults on Instruction Op-Code. They can be either permanent faults when the code memory is corrupted, or transient faults when an element of the micro-architecture like the instruction buffer is attacked. The case of permanent VS transient faults is discussed in Section 5.6.3. Changing one instruction into another in the general sense is a hard problem in symbolic execution as it relates to self-modifying code, inducing a state explosion to consider all possible options for each instruction. We do not have an efficient forkless algorithm for faulting instruction op-codes. However, some fault models can be considered as specific cases of instruction corruption like the test inversion fault model where a conditional jump (jump-if-equal for instance) is transformed into its opposite (jump-if-not-equal in that example). Instruction skips can also be seen as a specific case of instruction corruption, changing an instruction to a NOP.

Combination of Fault Models. BINSEC/ASE does not currently support the possibility to combine fault models in one multi-fault adversarial path. It would make sense in a security scenario considering an advanced attacker able to leverage different attack vectors in one attack. We have not imagined a more efficient algorithm than a naive combination of fault models into one big encoding. This naive technique could be implemented without much effort and we leave as future work to design an efficient combined injection technique.

Generic Support for Fault Models. We acknowledge that we implemented only a limited number of fault models having specific effects in this prototype. The aim was

to showcase the ability of our tool to support various fault models that can represent existing security scenarios. The interested user can extend BINSEC/ASE with new fault models more suitable for their analysis context, based on our developer guide (Section 5.5).

5.6.2 Faults on Intermediate Representation

Working on intermediate representation allows to abstract from specific ISAs and to have a generic analysis. The semantics are preserved, however, it changes the syntax. This becomes meaningful when considering faults modifying the binary encoding of instructions, for instance flipping a bit in the op-code or switching one register for another. Those faults require micro-architectural details to be known to the analysis, at least through a mapping of ISA op-codes to DBA instructions. This is another difficulty hindering the support of generic faults on instructions by BINSEC/ASE. Only restrictions that can be expressed with DBA and that can be reasoned about without micro-architectural information (instruction skip, test inversion, operand modification) have been implemented. We also miss micro-architectural considerations such as fetch and prefetch of instructions, without which we cannot model faults happening in those components.

5.6.3 Permanent VS Transient Faults

In this work, we only consider transient faults, i.e. a corruption that lasts until the value is overwritten. We believe permanent faults require micro-architectural information to precise exactly which bit of which physical component is affected, and then propagate it to the ISA level. Moreover, memory is often encrypted and permanent damage in it can be hard to reason with. Some restrictions on permanent faults can likely be implemented, such as permanent faults on a register, which is available at DBA level.

Chapter 6

Experimental Evaluation

Contents

This chapter is dedicated to the experiment evaluation of the approach presented in this thesis through the BINSEC/ASE prototype described in Chapter 5. After presenting our experimental setting (Section 6.1), we confirm the correctness and k-completeness of our BINSEC/ASE prototype (Section 6.2) and systematically evaluate its performance against our implementation of the forking technique representing the state-of-the-art approach and compared to our various optimizations for arbitrary data faults (Section 6.3) and for the other faults models (Section 6.4). We also investigate the advantages of using a forkless fault encoding in instrumentation (Section 6.5). Secondly, we explore various security scenarios (Section 6.6) and a case study (Section 6.7) to showcase the interest and feasibility of our technique.

6.1 Evaluation Overview

In this section, we introduce the research questions driving this experimental evaluation of our technique, FASE (Forkless Adversarial Symbolic Execution), implemented in the BINSEC/ASE tool. We also detail the experimental settings and present the benchmarks used for the evaluation.

6.1.1 Research Questions

We aim to answer the following research questions during this evaluation.

RQ 1: Correctness and K-completeness. FASE algorithm has been shown to be correct and k-complete for adversarial reachability (see Section 4.3.3.3). We start by checking that our FASE implementation preserves those properties in Section 6.2.

RQ 2: FASE Evaluation for Arbitrary Data Faults. In Section 6.3, we evaluate the performance of our technique compared to the forking technique implemented alongside FASE in BINSEC/ASE. We also compare the performance of our optimizations and forkless encoding variations through the following research questions.

- **RQ 2.1: Scalability.** We verify that FASE is able to scale in number of faults without path explosion, better than the forking technique (Section 6.3.1);

- **RQ 2.2: Impact of Optimizations.** We investigate the impact of our optimizations on the performance of FASE and wonder which is the fastest optimization (Section 6.3.2);
- **RQ 2.3: Forkless Encodings.** We investigate the impact of the operators used for the forkless encoding and wonder which one does result in the fastest analysis (Section 6.3.3).

RQ 3: FASE Evaluation for Other Fault Models. We evaluate the impact of the other implemented fault models: reset (RQ3.1 Section 6.4.1), bit-flip (RQ3.2 Section 6.4.2), test inversion (RQ3.3 Section 6.4.3) and instruction skip (RQ3.4 Section 6.4.4). For each, we verify that we can scale in number of faults without path explosion, compared to the forking technique.

RQ 4: Forkless Faults in Instrumentation. In Section 6.5, we evaluate the performance of forkless encodings compared with forking ones in source-level instrumentation. The instrumentation process is described in Appendix C. We check that we can scale in number of faults without path explosion, compared to the forking technique in instrumentation.

6.1.2 Experimental Setting

We now present the benchmark used for this performance evaluation and detail our experimental settings.

6.1.2.1 Benchmark

Performance Benchmarks. We used here a standard set of programs from the SWIFI literature on physical fault injections and high-security devices. This constitutes a benchmark of 12 programs, characterized in Table 6.1.

- Eight versions of VerifyPIN from the FISSC [DPP+16] benchmark suite, dedicated to the evaluation of physical fault attack analyses. VerifyPIN is a toy authentication program. There are one unprotected and 7 different protected versions, some vulnerable, some resistant to one test inversion fault. An oracle is provided by FISSC, checking if the user PIN truly corresponds to the reference PIN;
- Two manually unrolled versions of the unprotected VerifyPIN, with a PIN size of 4 and 16, to add diversity in the benchmarks with programs without loops;
- Two versions of the npo2 program from Le *et al.* [LHGD18], together with their oracles. Npo2 is a program computing an integer's upper power of two. The attacker's goal is to perform a silent data corruption, i.e. change the end result without triggering countermeasures. One version is vulnerable to one arbitrary data fault, the second is resistant due to extra arithmetic checks.

In the remainder of this chapter, this set of programs is denoted as the *performance benchmarks*.

Compilation. The benchmarks are written in C and have been compiled with gcc for the Intel x86-32 architecture. This was the first ISA supported by BINSEC/ASE and we expect other ISAs to give the same results as only the lifting of the ISA to DBA changes, not the adversarial symbolic execution itself. We use the flag "-O0" to preserve countermeasures, and subsequently to improve assembly readability. For BINSEC compatibility, we use the "-static" flag to include the necessary library functions directly in

Table 6.1: Benchmarks characteristics and statistics of a standard SE analysis

			BINSEC analysis - no fault			
Program group (#)	C loc	x86 loc	#instructions (explored)	#paths	max #branch in a path	Time
Sections 6.2 to 6.5						
VerifyPINs (8)	80-140	160-215	192-269	1	17-34	< 0.1s
VerifyPIN unrolled (2)	40-85	140-430	142-442	5-17	5-17	< 0.1s
npo2 (2)	50	200-220	607-653	3	31-33	< 0.1s

the binary.

6.1.2.2 Attacker Model

The attacker model chosen in this evaluation can perform a varying number of faults. Its goal is expressed as a security oracle directly written in C for each benchmark, the computation of which is not faulted. Specific details are provided for the different experiments when relevant. In particular, the exact fault models used are indicated. Except when explicitly stated otherwise, we do not fault addresses, nor fault directly the stack pointer or the program counter, and we do not fault the status flags.

6.1.2.3 Competitor

We chose to represent the state-of-the-art work with the forking technique, which scales better than mutant generation techniques for multiple faults. The forking technique split the explored path into two at each possible fault location, creating one containing the fault, and keeping one without the fault at that particular location. In order to only compare encodings we implemented forking faults encodings inside BINSEC/ASE, along with our new forkless ones, as described in Chapter 5. Note that the forking technique induces path splits in the analysis exploration at fault locations, it does not explicitly fork processes. We compare our approach to others on a more general level in Section 4.5.

6.1.2.4 Experimental Setup

Machine Used. We ran our experiments on a cloud machine with a processor Intel Dual Xeon 4214R with 48 CPU cores and 384GB of RAM. Experiments ran in parallel on the 48 cores, each run using only one core.

BINSEC settings. We limit the maximal depth of an analysis to the depth necessary to perform an exhaustive non-faulty analysis, rounded to the upper hundred. We may not always be complete, which is not necessary for a performance evaluation. We exhaustively explore all the possible adversarial paths up to this bound in order to have comparable results. We set the global analysis timeout for 1 day.

6.1.3 Artifact Availability

Our benchmark infrastructure, case studies and the executable of BINSEC/ASE have been made available through artifacts for reproducibility purposes on GitHub[1] and Zenodo[2]. The BINSEC/ASE source code will be open-sourced.

6.2 Correctness and K-completeness (RQ1)

We start with a basic sanity check, in order to check whether our implementation indeed is correct and k-complete.

RQ1. Is our tool correct and k-complete?

Goal. The goal of this first research question is to check that our tool works as expected on several codes with known ground truth. In particular, we show that we are able to find attacks on vulnerable programs and provide a complete analysis showing the absence of vulnerabilities in secure programs.

Protocol. The performance benchmarks are used in this experiment. We consider an attacker model able to perform either 1 or 2 arbitrary data faults, or 1 or 2 test inversion faults. To study the correctness and completeness of BINSEC/ASE, we proceed as follows.

1. We check that indeed, with no fault allowed, no attack is found in any of the benchmarks. We make sure that the analysis is complete for this setting;
2. For each program of the benchmark, we get the ground truth provided by their authors, setup the same attacker model, and observe BINSEC/ASE output.

Table 6.2: FASE vulnerability results compared to benchmark ground truth (RQ1)

AD: arbitrary data, TI: test inversion

V: vulnerable, **S**: secure, _: unspecified in the reference paper

Program	Ground truth / FASE result			
	1 TI	2 TI	1 AD	2 AD
npo2 c1 (insecure)	_ / **S**	_ / **S**	*V* / *V*	*V* / *V*
npo2 c2 (secure)	_ / **S**	_ / **S**	**S** / **S**	**S** / **S**
VerifyPIN 0	*V* / *V*	*V* / *V*	_ / *V*	_ / *V*
VerifyPIN 1	*V* / *V*	*V* / *V*	_ / *V*	_ / *V*
VerifyPIN 2	*V* / *V*	*V* / *V*	_ / *V*	_ / *V*
VerifyPIN 3	*V* / *V*	*V* / *V*	_ / *V*	_ / *V*
VerifyPIN 4	*V* / *V*	*V* / *V*	_ / *V*	_ / *V*
VerifyPIN 5	**S** / **S**	*V* / *V*	_ / *V*	_ / *V*
VerifyPIN 6	**S** / **S**	*V* / *V*	_ / *V*	_ / *V*
VerifyPIN 7	**S** / **S**	*V* / *V*	_ / *V*	_ / *V*
VerifyPIN unrolled size 4	**S** / **S**	**S** / **S**	_ / *V*	_ / *V*
VerifyPIN unrolled size 16	**S** / **S**	**S** / **S**	_ / *V*	_ / *V*

Results. Results are summarized in Table 6.2. Vulnerability results are the same for FASE and its optimizations, as well as for the forking technique. The exploration of our

[1] https://github.com/binsec/esop2023_artefact
[2] https://zenodo.org/record/7507112

analysis was either complete or at least an adversarial path was found, demonstrating the vulnerability of the program. As expected, the insecure npo2 program is vulnerable to a single arbitrary data fault while the secure version is not, nor is it vulnerable to two arbitrary data faults. According to their authors, the VerifyPIN versions 0 to 4 are vulnerable to one test inversion, while VerifyPIN 5 to 7 are resistant to it. We indeed reproduce these results. When allowing two faults, all VerifyPINs become vulnerable. When using one arbitrary data fault against the VerifyPINs, all versions are found vulnerable. We manually check that indeed the identified attack paths make sense. Our manually unrolled versions of VerifyPINs do not contain conditional branching instructions in the targeted function, making them resistant to test inversion. We check that this is the case, while they are still vulnerable to a single arbitrary data fault.

Conclusion RQ1. Our implementation of FASE preserves its correctness and k-completeness properties on our benchmark. In particular, we can show a program's vulnerability to fault injection attacks and prove a program secure when it is.

6.3 FASE Evaluation for Arbitrary Data Faults (RQ2)

Arbitrary data faults are the most complex fault model BINSEC/ASE implements with regard to the number of added symbolic variables. One determines the activation of a fault and the second the corrupted value, and that for each possible fault location, encompassing all assignments. Hence, characterizing FASE for arbitrary data faults is a form of worst-case evaluation, and also where the efficiency gain is the most important.

We only consider arbitrary data faults in this section. First, we compare scaling capabilities of FASE against the forking technique in terms of number of faults. Then, we want to show which optimization results in the fastest analysis and what their impact on the analysis is. We also evaluate the impact of the forkless encoding operator used and show which is the fastest. We consider optimizations and encodings to be independent and evaluate them separately. As a reminder, the analyses are set to explore exhaustively all program paths within the 24-hour time limit.

6.3.1 Scalability (RQ2.1)

First, we show the general performance improvement of FASE over the forking technique, especially as more faults are considered. We selected our best performing optimization, FASE-IOD, as shown in Section 6.3.2, to represent FASE.

RQ2.1. Can we scale in number of faults without path explosion compared to the forking technique (for arbitrary data faults)?

Goal. The goal of this experiment is to check whether our technique is able to prevent path explosion, especially when the number of faults increases, and that it translates to improved analysis time.

Protocol. The performance benchmark is used in this experiment. We consider an attacker model able to perform 1 to 10 arbitrary data faults.

Results. We start by looking at the analysis time to compare FASE and the forking technique, as analysis time conditions the usability of a tool. Both techniques have the same exploration strategy, analyze all possible paths, and can be compared fairly.

Analysis time results are presented in Table 6.3 (p.109) and illustrated in Figure 6.1 (p.109). First, we see that the forking technique starts to timeout for half of the benchmarks for an attacker model able to perform 3 faults, and timeout for all for 8 faults and above. FASE never timeouts in this experiment. FASE is on average x19 times faster for 1 fault and x403 times faster for 2 faults. Values for 3 faults and above for the forking technique are for incomplete runs, hence not comparable.

Table 6.3: FASE-IOD and forking analysis time comparison for arbitrary data faults (RQ2.1)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
average value							
FASE-IOD	2.08	8.88	33.4	100	467	909	1.04k
Forking*	39.5	3.58k	8.17k	9.15k	35.6k	86.4k	86.4k
timeouts (24h) over 12 benchmarks in total							
FASE-IOD	0	0	0	0	0	0	0
Forking	0	0	6	9	11	12	12

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts)

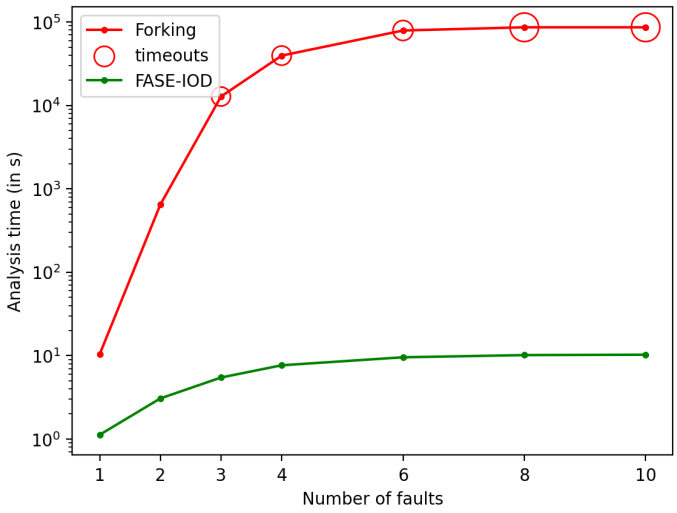


Figure 6.1: FASE-IOD and forking analysis time comparison for arbitrary data faults (RQ2.1)

Then, we compare both techniques on the number of explored paths. FASE aims to reduce path explosion, we check whether it is experimentally true. The number of explored paths is presented in Table 6.4 (p.110) and illustrated in Figure 6.2 (p.110). The forking technique shows an exponential increase in the number of paths explored as the number of faults increases, while FASE increase in explored paths is much more mitigated, still opening new paths as they become feasible due to the introduced faults. On average, the forking technique explores x17 times more paths for 1 fault and x267 times more for 2 faults.

Lastly, we notice that values from around 4 faults to 10 tend to plateau. This is especially visible in Figures 6.1 (p.109) and 6.2 (p.110). This is due not to our technique but to the fact that the programs in our benchmark are not overly complex, and 4 to

Table 6.4: FASE-IOD and forking number of explored paths comparison for arbitrary data faults (RQ2.1)

	Number of explored paths						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE-IOD	22.3	108	381	1.05k	4.64k	11.1k	15.2k
Forking*	369	28.8k	170k	215k	1.41M	2.17M	2.2M

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts)

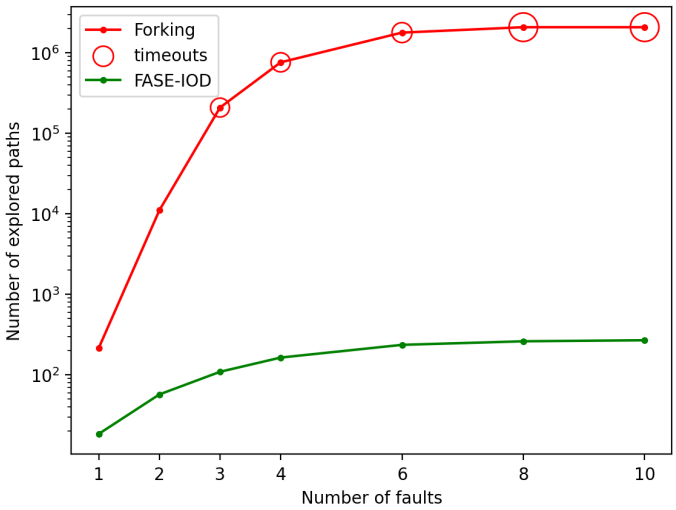


Figure 6.2: FASE-IOD and forking number of explored paths comparison for arbitrary data faults (RQ2.1)

6 faults are sufficient to open all possible paths in the control-flow graph. Hence the number of explored paths plateaus. Interestingly, the analysis time also plateaus, as adding unnecessary faults does not open new paths to explore and relax the constraints on fault placing, not making the solver's tasks harder.

> **Conclusion RQ2.1.** For arbitrary data faults, FASE is able to scale in the number of faults considered while the forking technique struggles. A direct correlation between the path explosion experienced by the forking technique and its slower analysis time can be observed, compared to FASE which mitigates path explosion and obtains faster analysis times.

6.3.2 Impact of Optimizations (RQ2.2)

We now evaluate our different optimizations for FASE. Two angles were considered to optimize FASE analysis:

- We have two optimizations changing the computation of the path predicates while keeping the same attacker power, EDS[3] and IOD[4], plus their combination EDS+IOD;

[3]Early Detection of fault Saturation
[4]Injection On Demand

– In addition, we also implemented optimizations aimed at adding query simplifications between their computation and them being sent to the solver, sub-fault simplification (SFS), described in Section 5.3.6. As a reminder, SFS aims to detect and nullify faults subsumed by another one. Different variants were implemented.

RQ2.2. What is the impact of our optimizations?

Goal. The goal of this experiment is to evaluate our optimizations and how they contribute to reducing query complexity.

Protocol. The performance benchmark is used in this experiment. We consider first an attacker model able to perform 1 to 10 arbitrary data faults for our first optimizations (EDS, IOD and EDS+IOD) evaluation. Then, for SFS, we consider an attacker able to inject 1 to 4 arbitrary data faults.

Results - EDS, IOD, EDS+IOD. We start with our first set of optimizations and look at whether they result in a faster analysis, with all parameters equal otherwise. Analysis time data is presented in Table 6.5 (p.111) and illustrated in Figure 6.3 (p.112). Overall, as expected, FASE without optimizations is the slowest, followed by FASE-EDS (x1.8 times faster than FASE for 1 fault), then FASE-EDS+IOD (x3.3 times faster than FASE for 1 fault) and the fastest being FASE-IOD (x3.7 times faster than FASE for 1 fault).

Table 6.5: FASE optimizations, analysis time for arbitrary data faults (RQ2.2)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
average value							
FASE	7.73	27.6	92.6	261	1.06k	2.2k	2.56k
FASE-EDS	4.27	17.6	64.0	201	958	2.22k	2.7k
FASE-IOD	2.08	8.88	33.4	100	467	909	1.04k
FASE-EDS+IOD	2.36	10.5	38.5	111	502	936	1.01k
Forking*	39.5	3.58k	8.17k	9.15k	35.6k	86.4k	86.4k
timeouts (24h) over 12 benchmarks in total							
FASE	0	0	0	0	0	0	0
FASE-EDS	0	0	0	0	0	0	0
FASE-IOD	0	0	0	0	0	0	0
FASE-EDS+IOD	0	0	0	0	0	0	0
Forking	0	0	6	9	11	12	12

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts)

We continue our experimental evaluation with a sanity check, looking at the number of explored paths, presented in Table 6.6 (p.112). FASE, FASE-EDS, FASE-IOD and FASE-EDS+IOD explore the same number of paths, which is expected since they encode the same attacker model.

To provide more intuition on how optimizations impact the analysis and in particular contribute to reducing query complexity, we consider other, internal, metrics. First, we consider the ratio of queries created over queries sent to the solver, presented in Table 6.7 (p.113), to see if some queries have been made so simple that they can be arithmetically reduced to true or false by the analysis, without the help of the

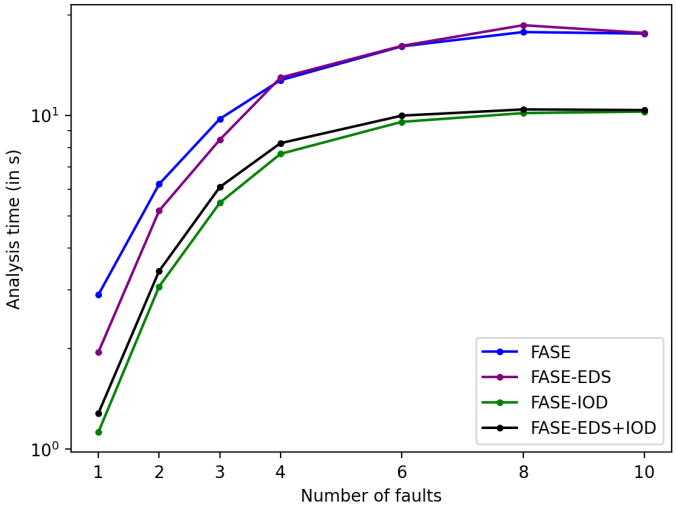


Figure 6.3: FASE optimizations, analysis time for arbitrary data faults (RQ2.2)

Table 6.6: FASE optimizations, number of explored paths for arbitrary data faults (RQ2.2)

	Number of explored paths						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE	22.3	108	381	1.05k	4.64k	11.1k	15.2k
FASE-EDS	22.3	108	381	1.05k	4.64k	11.1k	15.2k
FASE-IOD	22.3	108	381	1.05k	4.64k	11.1k	15.2k
FASE-EDS+IOD	22.3	108	381	1.05k	4.64k	11.1k	15.2k
Forking*	369	28.8k	170k	215k	1.41M	2.17M	2.2M

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts)

solver. At first glance, we notice the optimizations generate slightly more queries than FASE, as expected from the related algorithms. For instance, on average for 1 fault compared to FASE, FASE-EDS and FASE-IOD generate x1.04 times more queries and FASE-EDS+IOD generates x1.09 times more queries. Despite this slight increase, fewer queries are sent to the solver with the optimizations. For instance, for 1 fault, 44% of queries are simplified without a solver call for FASE, 58% for FASE-EDS, 65% for FASE-IOD and 62% for FASE-EDS+IOD. This shows that generating more queries can, somewhat counterintuitively, reduce overall query complexity, as up to 2/3 are resolved before needing to be sent to the solver. For the sake of comparison, 51% of queries are simplified without a solver call for the forking technique.

The second internal metric measuring query complexity is the average solving time per query, presented in Table 6.8 (p.114) and illustrated in Figure 6.4 (p.114). Measures for average solving time per query tend to vary somewhat, which can have a significant impact due to the small values. To mitigate this, we perform 100 runs for each configuration (program, number of faults, technique) and only use the average values per program. FASE encodings introduce many new symbolic variables and are expected to increase the average solving time per query, which indeed is more than twice the value of the forking technique (x2.7 times averaging values for 1 and 2 faults). As expected, our optimizations reduce the average solving time per query. On average for all num-

Table 6.7: FASE optimizations, number of queries created and sent to the solver for arbitrary data faults (RQ2.2)

	Number of queries created / Number of queries sent to the solver					
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE	295 / 164	1.24k / 679	3.99k / 2.11k	10.1k / 5.22k	37.1k / 18.7k	69.7k / 34.2k	83.1k / 39.8k
FASE-EDS	308 / 130	1.32k / 584	4.32k / 1.98k	11.1k / 5.29k	41.8k / 20.9k	79.6k / 39.9k	94.0k / 45.4k
FASE-IOD	308 / 109	1.38k / 558	4.62k / 1.98k	12.1k / 5.46k	46.5k / 22.9k	89.0k / 44.9k	106k / 53.0k
FASE-EDS+IOD	321 / 121	1.51k / 650	5.02k / 2.25k	13.1k / 6.14k	50.1k / 25.2k	96.6k / 49.3k	116k / 58.0k
Forking*	3.54k / 1.74k	204k / 128k	1.03M / 533k	1.5M / 557k	6.99M / 4.44M	7.53M / 4.76M	7.02M / 4.65M

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts)

bers of faults, FASE-EDS is x1.04 faster than FASE in solving time, and FASE-IOD and FASE-EDS+IOD are x2 times faster than FASE.

Table 6.8: FASE optimizations, average solving time per query for arbitrary data faults (RQ2.2)

	Average solving time per query (s)						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE	0.050	0.041	0.037	0.037	0.038	0.041	0.038
FASE-EDS	0.039	0.045	0.037	0.037	0.036	0.041	0.037
FASE-IOD	0.020	0.020	0.021	0.021	0.019	0.020	0.019
FASE-EDS+IOD	0.020	0.019	0.021	0.021	0.020	0.020	0.019
Forking*	0.015	0.018	0.012	0.012	0.008	0.019	0.020

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts)

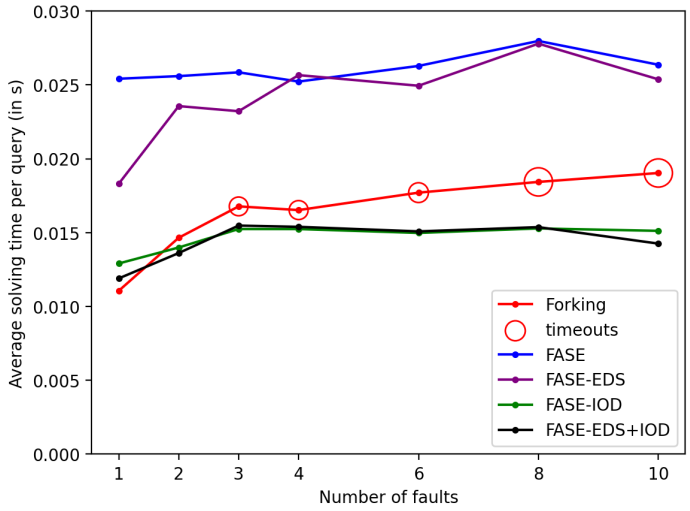


Figure 6.4: FASE optimizations, average solving time per query for arbitrary data faults (RQ2.2)

Our last internal metric to measure query complexity is the average number of ite operators per query, presented in Table 6.9 (p.115). For this metric, we add the values for non-faulted analysis for comparison. For a non-faulted run, there is almost no ite operator in queries, which makes an average of 0. FASE queries contain on average x7.4 times more ite operators per query for 1 fault than the forking technique. This value continues to increase as we consider more faults, but only slightly, as faulted expressions are already computed in the path predicate, they are simply less constrained in number of activated faults. The slight increase is due to some new paths opening thanks to faults. FASE's optimizations follow the same trend as FASE. On average for all fault numbers, FASE-EDS only spear 6% of ite operators compared to FASE, while FASE-IOD spears 39% of ite operators and FASE-EDS+IOD 38%. This shows that our optimizations can indeed reduce injection points in query. Interestingly, the forking technique also induces more ite operators in queries as the number of faults considered increases, likely due to the new possibility offered by faults in conditional jumps that

cannot be arithmetically simplified anymore, in particular for the arbitrary data faults in this experiment.

Table 6.9: FASE optimizations average number of ite operator per query for arbitrary data faults (RQ2.2)

	Average number of ite operator per query							
	0f	1f	2f	3f	4f	6f	8f	10f
average value								
FASE	0	2.87k	2.94k	2.99k	3.02k	3.05k	3.06k	3.07k
FASE-EDS	0	1.64k	3.11k	2.95k	3.0k	3.05k	3.06k	3.07k
FASE-IOD	0	1.53k	1.76k	1.83k	1.88k	1.92k	1.94k	1.94k
FASE-EDS+IOD	0	1.53k	1.94k	1.86k	1.9k	1.93k	1.94k	1.94k
Forking*	0	368	570	647	754	1.24k	1.32k	1.37k

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts)

Finally, we remark that a plateau appears in the different metrics as 4 to 6 faults are sufficient to open all possible paths (detailed in Section 6.3.1), except for the average time per query, which does not directly depend on the number of paths explored.

Results - SFS. We start by looking at the analysis time, presented in Table 6.10 (p.115), to check whether SFS optimizations result in a faster analysis. Unfortunately, SFS variants yield significantly slower analysis times, which start to equal those of FASE for large numbers of faults.

Table 6.10: Analysis time for sub-fault simplification for arbitrary data faults (RQ2.2)

	Analysis time (s)			
	1f	2f	3f	4f
average value				
FASE	5.41	23.0	79.9	223
FASE-Free-Lazy	8.23	31.3	94.7	255
FASE-Free-Abort	6.77	25.9	85.1	225
FASE-Mem-Lazy	17.2	44.2	116	215
FASE-Mem-Abort	16.0	40.3	106	217

To get more intuitions on the impact of SFS optimizations, we consider the average solving time per query. As expected, SFS allows for slower solving time, as it computes query simplifications. SFS reduces the average solving time per query, from x1.1 to x1.4 times faster. The memoization of part of the queries does not yield faster processing than traversing queries each time (FASE-Free). Aborting SFS mechanism proves more efficient than risking extra queries with the lazy approach.

SFS only adds a constraint on queries stating that the selected faults are null to help the solver. BINSEC/ASE does not implement arithmetical simplification rules complex enough to be at query scale, hence SFS does not reduce the number of ite operators in queries as shown in Table 6.12.

Table 6.11: Average solving time per query for sub-fault simplification for arbitrary data faults (RQ2.2)

Average solving time per query (s)				
	1f	2f	3f	4f
average value				
FASE	0.030	0.025	0.026	0.027
FASE-Free-Lazy	0.026	0.020	0.020	0.020
FASE-Free-Abort	0.028	0.023	0.024	0.025
FASE-Mem-Lazy	0.026	0.020	0.019	0.019
FASE-Mem-Abort	0.028	0.024	0.024	0.024

Table 6.12: Average number of ite per query for sub-fault simplification for arbitrary data faults (RQ2.2)

Average number of ite per query				
	1f	2f	3f	4f
average value				
FASE	2.87k	2.94k	2.99k	3.02k
FASE-Free-Lazy	2.87k	2.94k	2.99k	3.02k
FASE-Free-Abort	2.87k	2.94k	2.99k	3.02k
FASE-Mem-Lazy	2.87k	2.94k	2.99k	3.02k
FASE-Mem-Abort	2.87k	2.94k	2.99k	3.02k

Conclusion RQ2.2. Our different optimizations reduce query complexity compared to FASE. While EDS, IOD and EDS+IOD result in significantly faster analysis time overall, SFS slows down the analysis, its computation inside BINSEC is likely too expensive compared to the gain obtained in query complexity. FASE-IOD is our best-performing optimization. It is followed by FASE-EDS+IOD where the combination of optimizations does not produce better results, maybe due to the redundancy between the under-approximation counter of IOD and the saturation logic of EDS. Then, FASE-EDS only slightly improves upon FASE. Last, SFS while simplifying queries results in worst performance.

6.3.3 Comparison of the Different Forkless Encodings (RQ2.3)

We now consider different forkless encodings with various operators as described in Section 5.3.3.1 (reminded Table 6.13) for the arbitrary data fault model. We compare four forkless arbitrary data encodings, FASE-Ite, FASE-Mul (with a multiplication operator), FASE-And (with a logical and operator) and FASE-Xor (with a logical xor operator). We used FASE-Ite until now, as it will be shown to be the fastest encoding. Here, FASE is used without optimizations.

RQ2.3. What is the impact of the different encodings on the performance of the solver?

Goal. Our goal is to explore the impact of different operators in the forkless encoding and their affinity with the solver used by BINSEC/ASE.

Protocol. The performance benchmarks are used in this experiment. We consider an attacker model able to perform 1 to 10 arbitrary data faults.

Table 6.13: Arbitrary data encodings and their associated activation constraint

Fault model	Fault expression	Activation constraint
None	$x := y$	
Inlined if-then-else	$x := ite\ b_{addr}\ ?\ non\_det_{addr}\ :\ y$	$(non\_det_{addr} \neq y)\ \|\|\ \neg b_{addr}$
Multiplication	$x := y + b_{addr}\ \times non\_det_{addr}$	$(non\_det_{addr} \neq 0)\ \|\|\ \neg b_{addr}$
Bitwise and	$x := y + (-b_{addr})\ \&\ non\_det_{addr}$	$(non\_det_{addr} \neq 0)\ \|\|\ \neg b_{addr}$
Bitwise xor	$x := y \oplus (-b_{addr})\ \&\ non\_det_{addr}$	$(non\_det_{addr} \neq 0)\ \|\|\ \neg b_{addr}$

Results. We start by looking at the impact of encodings on the general analysis time, presented in Table 6.14 (p.117). For 1 fault, FASE-Ite is x1.2 times faster than FASE-Xor, and x1.5 times faster than FASE-Mul and FASE-And. FASE-Xor starts to perform better than FASE-Ite for 8 faults and more. FASE-Mul and FASE-And start to experiment timeouts at 8 and 6 faults respectively.

Table 6.14: Encodings, analysis time for arbitrary data faults (RQ2.3)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
average value							
FASE-Ite	8.59	27.7	96.9	271	1.13k	2.18k	2.63k
FASE-Xor	10.4	31.0	105	287	1.16k	1.77k	2.21k
FASE-Mul*	12.5	46.4	213	884	6.75k	7.32k	7.32k
FASE-And*	12.5	52.7	308	1.74k	7.31k	7.31k	7.31k
timeouts (24h) over 12 benchmarks in total							
FASE-Ite	0	0	0	0	0	0	0
FASE-Xor	0	0	0	0	0	0	0
FASE-Mul	0	0	0	0	0	1	1
FASE-And	0	0	0	0	1	1	1

FASE-Mul*: values for 8 faults and above are computed for incomplete runs due to timeouts,
FASE-And*: values for 6 faults and above are computed for incomplete runs due to timeouts

Then, we evaluate the encodings on internal metrics, evaluating query complexity. The ratio of queries created on the number of queries sent, presented in Table 6.15 (p.118), to reflect query complexity before solver call. The number of queries created and sent ratio is the same for the 4 encodings up to 4 faults, around 50%. FASE-Mul and FASE-And start to experience timeouts for 6 faults and above. This shows that arithmetic simplification rules implemented inside BINSEC to simplify expressions before queries are sent to the solver impact our four operators in the same way.

Our last query complexity metric is the average solving time per query, presented in Table 6.16 (p.118), to reflect the query complexity in solver calls. FASE-Ite is the fastest, followed by FASE-Xor x1.3 times slower on average for all number of faults, FASE-Mul is x2.4 times slower and FASE-And x3 times.

Table 6.15: Encodings, number of queries created and sent to the solver for arbitrary data faults (RQ2.3)

Number of queries created / Number of queries sent to the solver				
	1f	2f	3f	4f
average value				
FASE-Ite	295 /164	1.24k /679	3.99k /2.11k	10.1k /5.22k
FASE-Xor	295 /164	1.24k /679	3.99k /2.11k	10.1k /5.22k
FASE-Mul	295 /164	1.24k /679	3.99k /2.11k	10.1k /5.22k
FASE-And	295 /164	1.24k /679	3.99k /2.11k	10.1k /5.22k

	6f	8f	10f
FASE-Ite	37.1k /18.7k	69.7k /34.2k	83.1k /39.8k
FASE-Xor	37.1k /18.7k	69.7k /34.2k	83.1k /39.8k
FASE-Mul*	37.1k /18.7k	17.3k /9.22k	19.6k /10.2k
FASE-And*	10.9k /5.79k	11.9k /6.16k	16.9k /8.67k

FASE-Mul*: values for 8 faults and above are computed for incomplete runs due to timeouts,
FASE-And*: values for 6 faults and above are computed for incomplete runs due to timeouts

Table 6.16: Encodings, average solving time per query for arbitrary data faults (RQ2.3)

Average solving time per query (s)							
	1f	2f	3f	4f	6f	8f	10f
average value							
FASE-Ite	0.056	0.042	0.048	0.042	0.041	0.043	0.042
FASE-Xor	0.094	0.049	0.054	0.052	0.049	0.060	0.047
FASE-Mul*	0.089	0.070	0.107	0.084	0.101	0.165	0.138
FASE-And*	0.088	0.091	0.092	0.128	0.207	0.201	0.149

FASE-Mul*: values for 8 faults and above are computed for incomplete runs due to timeouts,
FASE-And*: values for 6 faults and above are computed for incomplete runs due to timeouts

Conclusion. FASE-Ite is the best-performing encoding for arbitrary data faults with the solver used by BINSEC/ASE, up to 6 faults, then FASE-Xor becomes better. Variations in overall analysis time when changing the operator of the forkless encoding are significant. Arithmetic simplification rules implemented in BINSEC to simplify queries before sending them to the solver are also important and can spear solver calls if efficient. Here around 50% of queries are speared.

In this experiment, we only considered the solver natively bound with BINSEC/ASE, Bitwuzla. However, different solvers may favor different operators. We investigate this question in Appendix B.5, comparing Bitwuzla to two other common solvers. While Boolector also favors the 'ite' operator, z3 favors the xor encoding.

6.4 FASE Evaluation of Other Fault Models (RQ3)

After BINSEC/ASE evaluation for arbitrary data faults, we evaluate it for the other implemented fault models: reset, bit-flip, test inversion and instruction skip.

We aim to assess the extensibility of arbitrary data fault results. We expect the same performance trends for analysis time and the number of explored paths for other fault models, in particular for data faults since they are particular cases of arbitrary data faults.

RQ3. For each fault model, can we scale in number of faults without path explosion compared to the forking technique?

Protocol. We used the performance benchmarks in this experiment. We consider an attacker model able to perform 1 to 10 faults of the selected fault model. Our analysis explores exhaustively all possible paths until the time limit to obtain a fair comparison.

For data faults (reset and bit-flip) and for instruction skip, we only consider FASE-IOD as it is implemented for those fault models and this optimization was shown to be the best-performing one for arbitrary data faults. The test inversion fault model has no optimization implemented.

6.4.1 FASE Evaluation of Reset Faults (RQ3.1)

First, we consider the overall analysis time, presented in Table 6.17 (p.119) . FASE-IOD shows better performance than the forking technique for reset faults, on average we are x2.5 times faster for 1 fault, x16 times faster for 2 faults, x90 times faster for 3 faults and x560 times faster for 4 faults. The forking technique starts to timeout for 6 faults instead of 3 for arbitrary data faults. This is likely due to the simpler fault model, as the effect of the fault is hardcoded to zero, instead of being a new symbolic value.

Table 6.17: Analysis time for reset faults (RQ3.1)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
average value							
FASE-IOD	0.363	1.15	2.81	5.04	9.78	13.3	15.2
Forking	0.922	18.8	255	2.82k	3.69k	27.8k	11.3k
timeouts (24h) over 12 benchmarks in total							
FASE-IOD	0	0	0	0	0	0	0
Forking*	0	0	0	0	9	9	11

(Forking*: values for 6 faults and above are computed for incomplete runs due to timeouts)

Then, the number of explored paths results are presented in Table 6.18 (p.120). We see that the forking technique explodes in number of paths explored while FASE-IOD does not. On average, we explore x5.4 times fewer paths for 1 fault, x48 for 2 faults and x400 for 3 faults.

Conclusion RQ3.1. We confirmed for the reset fault model the general trends of improved performance for analysis time and number of explored paths observed so far for arbitrary data faults.

Table 6.18: Number of explored paths for reset faults (RQ3.1)

	Number of explored paths						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE-IOD	11.0	30.2	66.1	113	202	265	300
Forking*	59.2	1.45k	26.3k	388k	851k	8.48M	2.0M

(Forking*: values for 6 faults and above are computed for incomplete runs due to timeouts)

6.4.2 FASE Evaluation of Bit-Flip Faults (RQ3.2)

Analysis time results are presented in Table 6.19 (p.120). The forking technique seems to favor this fault model compared to arbitrary data and reset fault models. The same analysis time trends from other data faults can be seen, except that the forking line drops below FASE-IOD for 1 fault. The forking technique starts to experience timeouts for 4 faults. The forking technique is x1.9 times faster for 1 fault compared to FASE-IOD but becomes x3.9 times slower than FASE-IOD for 2 faults. A possible explanation for the forking technique being faster for one fault is the chosen forkless encoding, which results in harder-to-solve queries than the arbitrary data faults (in average solving time per query), that it is supposed to be a particular case of. Furthermore, as described in Section 5.3.7, the forking technique is implemented such that a fork in the analysis happens between a path with a bit-flip fault in an assignment and a path without. A path is not created for each possible bit the fault could flip. This implementation may be considered an optimization of the forking technique for bit-flips.

Table 6.19: Analysis time for bit-flip faults (RQ3.2)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE-IOD	1.25	2.89	4.69	6.35	8.76	9.27	9.15
Forking*	0.645	15.1	225	952	2.13k	13.4k	9.47k
	timeouts (24h) over 12 benchmarks in total						
FASE-IOD	0	0	0	0	0	0	0
Forking	0	0	0	2	7	8	10

(Forking*: values for 4 faults and above are computed for incomplete runs due to timeouts)

The number of explored paths are presented in Table 6.20 (p.121). We can see that the number of explored paths explodes for the forking technique, from 68.9 on average for 1 fault to 1.44k for 2. FASE-IOD is able to mitigate the path explosion experienced by the forking technique, which explores on average x4.8, x38 and x311 times more paths for 1, 2 and 3 faults respectively.

Conclusion RQ3.2. FASE-IOD outperforms the forking technique in scalability in number of faults considered. However, the forking technique results in a faster analysis time for 1 fault and then becomes much slower as more faults are considered. This could indicate that our forkless encoding of bit-flips is not the optimal one in terms of performance. We leave designing a faster one for future work.

Table 6.20: Number of explored paths for bit-flip faults (RQ3.2)

	Number of explored paths						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE-IOD	14.5	37.2	60.4	80.8	110	119	122
Forking	68.9	1.44k	18.8k	92.3k	207k	999k	734k

(Forking*: values for 4 faults and above are computed for incomplete runs due to timeouts)

6.4.3 FASE Evaluation of Test Inversion Faults (RQ3.3)

As a reminder, the test inversion fault model has no optimization implemented.

Analysis time results are presented in Table 6.21 (p.121). The forking technique results in faster analysis time for all numbers of faults, except for 10 faults. It is on average, over all number of faults, x1.4 times faster. Neither FASE nor the forking technique experiment timeouts for this fault model.

Table 6.21: Analysis time for test inversion faults (RQ3.3)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE	0.130	0.262	0.501	0.829	1.69	2.30	2.59
Forking	0.096	0.161	0.300	0.522	1.24	2.17	2.82
	timeouts (24h) over 12 benchmarks in total						
FASE	0	0	0	0	0	0	0
Forking	0	0	0	0	0	0	0

The number of explored paths is presented in Table 6.22 (p.121). The forking technique explores always more paths as the analysis considers more faults, but so does FASE, as by definition a test inversion fault opens a new path when activated. FASE explores slightly fewer paths on average. The gap in explored paths is significantly lesser than for the other presented fault models. There is not enough of a gain to compensate for the added query complexity of the forkless encoding. Furthermore, there are only few possible fault locations for test inversion in the performance benchmark, hence comparatively few new paths are created by the forking technique compared to data faults, which tips to our disadvantage the explored paths VS query complexity trade-off. However, as the gap in explored paths increases with faults, we can imagine that FASE, which is faster for 10 faults, remains faster for more than 10 faults with an increasing gain.

Table 6.22: Number of explored paths for test inversion faults (RQ3.3)

	Number of explored paths						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE	12.2	33.2	69.7	122	249	343	379
Forking	15.3	42.6	92.4	171	433	769	1.02k

Conclusion RQ3.3. FASE supports the test inversion fault model, however, it is not the fastest technique for a small number of faults. FASE is more interesting than the forking technique for a large number of faults (10 or more). This is likely because FASE tends to perform better than the forking technique when there is a big gap in the number of paths explored, which is the case for test inversion faults only for a large number of faults. We emit the hypothesis that faults at control-flow points would favor forking faults.

6.4.4 FASE Evaluation of Instruction Skip Faults (RQ3.4)

As a reminder, the only optimization implemented for instruction skips is IOD.

Analysis time results are presented in Table 6.23 (p.122). First, we can see that there is 1 program for which FASE-IOD cannot finish in time, the unrolled version of VerifyPIN with a PIN size of 16. It contains a long sequence of 175 assignments, that seem to overload the analysis when faulted in a forkless manner. For comparison, the basic VerifyPIN program never exceeds 10 consecutive assignments. We believe that in a program with regular branching instructions, constraints are regularly added, hence there is actually not that much freedom in variable assignments, contrary to a program with a long assignment sequence adding no constraints. The forkless instruction skip appears not well suited for programs with long assignment sequences.

The forking technique passes this program, likely because it spreads the difficulty of placing the fault by exploring different paths. The forkless technique does not experience any other timeout for the rest of the programs for any number of faults. The forking technique, however, starts to experience timeouts from 3 faults. With our incomplete results, FASE-IOD is slower for 1 fault but becomes faster for 2 faults. While exact numbers do not have meaning, since we count the full 24h in the case of a timeout, FASE-IOD is indeed faster for 2 faults onward.

Table 6.23: Analysis time for instruction skip faults (RQ3.4)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
average value							
FASE-IOD*	95.6	150	329	515	822	842	962
Forking*	9.6	857	3.52k	16.3k	32.1k	86.4k	86.4k
timeouts (24h) over 12 benchmarks in total							
FASE-IOD	1	1	1	1	1	1	1
Forking	0	0	3	6	10	12	12

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts
FASE*: all value from incomplete runs)

The number of explored paths results are presented in Table 6.24 (p.123). Despite the incomplete run, which is an unrolled program with very few paths to explore for FASE-IOD, we can see that FASE-IOD mitigated the path explosion faced by the forking technique for instruction skip faults.

Table 6.24: Number of explored paths for instruction skip faults (RQ3.4)

	Number of explored paths						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE-IOD*	19.1	87.7	283	734	2.67k	4.7k	5.78k
Forking*	164	11.0k	211k	3.44M	7.6M	28.1M	29.2M

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts
FASE*: all value from incomplete runs)

Conclusion. To conclude, instruction skip is a harder fault model for FASE-IOD, as we experience a timeout. However, FASE-IOD clearly out-performs the forking technique when considering multiple faults (3 faults or more).

6.4.5 Summary

Table 6.25 summarizes fault model data for comparison on analysis time, number of explored paths and average solving time per query.

Conclusion RQ3. FASE implements various fault models, with unequal results compared to the forking technique.

- We always outperform the forking technique for arbitrary data faults;
- We also always beat the forking technique for reset faults;
- It is only for 2 or more faults that FASE shows better results than the forking technique for bit-flip faults;
- We struggle more for instruction skip faults as we experience a timeout, but we clearly perform better than the forking technique for 3 or more faults.;
- FASE does not shine for test inversion faults as the forking technique experiences a restricted path explosion. We only outperform the forking technique for large numbers of faults (10 or more).

Overall, FASE tends to perform better than the forking technique for multi-fault analyses. It would seem like the forkless technique favors faults at data assignments, while the forking technique favors faults at control-flow points.

We refer the interested reader to Appendix B for more experimental data on fault models other than arbitrary data faults. In particular, we investigate the impact of the optimizations implemented for each fault model and comment on the impact of the chosen encoding operator for reset faults.

Table 6.25: Summary table comparing FASE and the forking technique (RQ3)

		1f	2f	3f	4f	6f	8f	10f
Analysis time (s)								
FASE-IOD	AD	2.08	8.88	33.4	100	467	909	1.04k
Forking		39.5	3.58k	8.17k*	9.15k*	35.6k*	86.4k*	86.4k*
FASE-IOD	RS	0.363	1.15	2.81	5.04	9.78	13.3	15.2
Forking		0.922	18.8	255	2.82k	3.69k*	27.8k*	11.3k*
FASE-IOD	BF	1.25	2.89	4.69	6.35	8.76	9.27	9.15
Forking		0.645	15.1	225	952*	2.13k*	13.4k*	9.47k*
FASE	TI	0.130	0.262	0.501	0.829	1.69	2.30	2.59
Forking		0.096	0.161	0.300	0.522	1.24	2.17	2.82
FASE-IOD	IS	95.6*	150*	329*	515*	822*	842*	962*
Forking		9.6	857	3.52k*	16.3k*	32.1k*	86.4k*	86.4k*
Explored paths								
FASE-IOD	AD	22.3	108	381	1.05k	4.64k	11.1k	15.2k
Forking		369	28.8k	170k*	215k*	1.41M*	2.17M*	2.2M*
FASE-IOD	RS	11.0	30.2	66.1	113	202	265	300
Forking		59.2	1.45k	26.3k	388k	851k*	8.48M*	2.0M*
FASE-IOD	BF	14.5	37.2	60.4	80.8	110	119	122
Forking		68.9	1.44k	18.8k	92.3k*	207k*	999k*	734k*
FASE	TI	12.2	33.2	69.7	122	249	343	379
Forking		15.3	42.6	92.4	171	433	769	1.02k
FASE-IOD	IS	19.1*	87.7*	283*	734*	2.67k*	4.7k*	5.78k*
Forking		164	11.0k	211k*	3.44M*	7.6M*	28.1M*	29.2M*
Average solving time per query (s)								
FASE-IOD	AD	0.020	0.020	0.021	0.021	0.019	0.020	0.019
Forking		0.015	0.018	0.012*	0.012*	0.008*	0.019*	0.020*
FASE-IOD	RS	0.005	0.006	0.007	0.006	0.006	0.007	0.006
Forking		0.003	0.003	0.003	0.003	0.001*	0.001*	0.002*
FASE-IOD	BF	0.035	0.024	0.024	0.024	0.026	0.026	0.027
Forking		0.003	0.004	0.003	0.002*	0.003*	0.004*	0.004*
FASE	TI	0.002	0.002	0.002	0.002	0.002	0.002	0.002
Forking		0.002	0.002	0.002	0.002	0.002	0.002	0.002
FASE-IOD	IS	0.017*	0.025*	0.023*	0.020*	0.023*	0.021*	0.023*
Forking		0.010	0.012	0.006*	0.005*	0.002*	0.003*	0.003*

XXX*: value computed for incomplete run due to timeouts
AD: arbitrary data, RS: reset, BF: bit-flip, TI: test inversion, IS: instruction skip

6.5 Forkless Faults in Instrumentation (RQ4)

Instrumentation is the process of adding data or metadata to a program, that an analyzer will pick up and reason upon. It is possible to inject faults through instrumentation in order to reuse existing analyzers, but it allows for less tuning of the analysis. For instance, the Lazart tool [PMPD14] takes C code as input and uses instrumentation with the Klee[5] symbolic execution engine. The goal of this section is to show that the

[5] https://klee.github.io/

forkless technique can also benefit instrumentation.

6.5.1 Experimental Settings

We start by briefly describing the experimental settings relevant to this section.

Benchmarks. We still use the performance benchmarks (see Section 6.1.2.1).

Instrumentation Process. Each source code, which is written in C, is instrumented with forkless and forking fault encodings. We can no longer use the BINSEC framework, easily at least, since we need a way to tell the analyzer which variables are symbolic and to specify constraints on them. We use the Klee symbolic engine instead and take advantage of its C-level API to instrument our C codes. We use Clang[6] to compile the instrumented C to bytecode, which is given as input to Klee for analysis.

To make sure the analysis would not fork for the forkless encoding, we did not use the ite operator, but the encoding using the bitwise and operator. For more details, we refer the interested reader to Appendix C.

Attacker Model. For this section, we consider an attacker model able to perform 1 to 10 arbitrary data faults.

Exploration Strategy. We set Klee to explore all paths for both the forkless and forking encoding, in order to be able to compare results.

Other Settings. We limit this experimental evaluation with a 1h timeout.

6.5.2 Scalability

RQ4. Can we scale in number of faults without path explosion, compared to the forking encoding when used in instrumentation?

Goal. The goal of this experiment is to check whether forkless faults can also benefit program instrumentation.

Results. We start by comparing the analysis time in order to see the interest of using the forkless encoding in instrumentation compared to the forking encoding. Analysis time results are presented in Table 6.26 (p.126) and illustrated in Figure 6.5 (p.126). Interestingly, the forking encoding is slightly faster (x1.07 times) for 1 fault than the forkless encoding. We believe results may vary depending on the operator used for the forkless encoding, as they did with BINSEC/ASE, we leave for future work to determine which forkless operator Klee favors and if one results in better performance than the forking encoding. There are also fewer injection points and paths to explore in C compared to binary code, which may tip the paths explored / query complexity trade-off in our disfavor. We see that, for a 1h timeout, the forkless technique also experiences a timeout when considering an attacker able to inject 3 faults. This happens for the VerifyPIN_7 program, the most complex of the VerifyPINs. The forking encoding also experiences a timeout for this program and 3 more when considering 3 faults. Overall, the forking encoding always experiences more timeouts than the forkless encoding. Except for 1 fault, the forkless encoding shows improved performance in analysis time (x6.4 times faster for 2 faults) and mitigates the path explosion experiences by the forking encoding.

[6] https://clang.llvm.org/

Table 6.26: Analysis time for arbitrary data faults in instrumentation with Klee (RQ4)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
average value							
Forkless*	14.6	92.9	350	505	820	1.1k	1.2k
Forking*	13.7	592	1.4k	1.93k	2.71k	2.8k	2.84k
timeouts (1h) over 12 benchmarks in total							
Forkless	0	0	1	1	2	3	4
Forking	0	0	4	5	8	10	10

(Forking*, Forkless*: values for 3 faults and above are computed for incomplete runs due to timeouts)

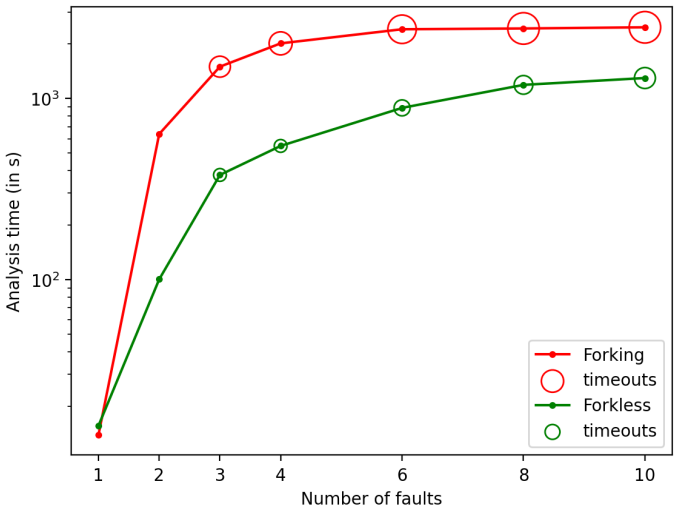


Figure 6.5: Forkless and forking analysis time for arbitrary data faults in instrumentation with Klee (RQ4)

We also consider the number of explored paths to compare instrumentation techniques, presented in Table 6.27 (p.127) and illustrated in Figure 6.6 (p.127). We can see that, like inside BINSEC, a binary symbolic execution engine, a forking encoding suffers a path explosion in instrumented C code when the number of faults increases. The forking encoding explores x2.3 times more paths when considering 1 fault and x8.7 times when considering 2.

We note that the same plateau effect as in the BINSEC/ASE evaluation appears, maybe for fewer faults even. This phenomenon is due in part to incomplete runs, and in part to the benchmarks used that are not so complex and few faults already open most paths of the CFG.

6.5.3 Conclusion

Table 6.27: Number of explored paths for arbitrary data faults in instrumentation with Klee (RQ4)

	Number of explored paths						
	1f	2f	3f	4f	6f	8f	10f
	average value						
Forkless*	17.7	84.3	154	384	420	498	527
Forking*	41.2	736	3.77k	15.0k	16.1k	12.5k	23.8k

(Forking*, Forkless*: values for 3 faults and above are computed for incomplete runs due to timeouts)

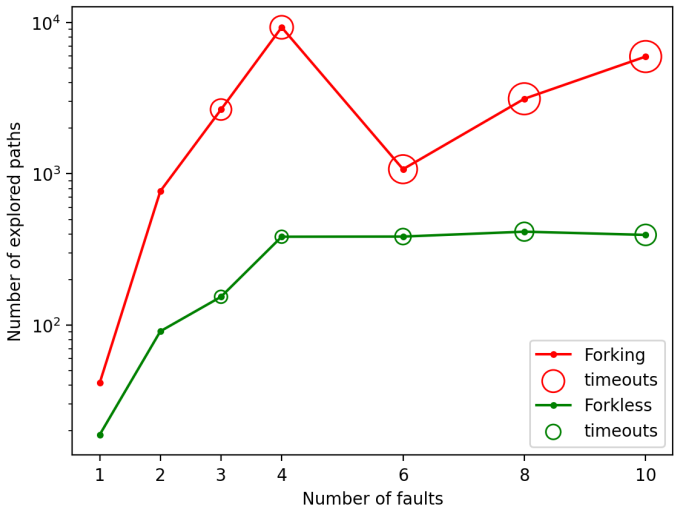


Figure 6.6: FASE and forking number of explored paths for arbitrary data faults in instrumentation with Klee (RQ4)

Conclusion RQ4. The forkless encoding enables improved performance and mitigates the path explosion experienced by the forking encoding when considering a multi-fault analysis, also when they are used for instrumentation, demonstrating the interest of the forkless encoding. Still, our other optimizations cannot be used.

6.6 Security Scenarios

In this section, we explore different security scenarios. This showcases the usability of our tool on various real-world programs, with different attacker models and attack goals. Programs' characteristics are presented in Table 6.28 (p.128).

In particular, we aim to show that:

- We can replay known attacks with BINSEC/ASE;
- Our technique is expressive enough to handle real-world scenarios;
- We are able to scale beyond our performance benchmarks to real-world programs.

Table 6.28: Benchmarks characteristics and statistics of a standard SE analysis

			BINSEC analysis - no fault			
Program group (#)	C loc	x86 or ARM loc	#instr (explored)	#paths	max #branch in a path	Time
Section 6.6						
CRT-RSA (3)	125-170	400-600	108k-29M	1	5k-1.3M	0.4s-1m27
Secret keeping machine (2)	100-200	240-360	1k-1.3k	1	130-150	< 0.1s
VerifyPIN_0 with SecSwift (1)	80	430	430	1	22	< 0.1s
Neural network (1)	100	275	774	1	109	0.1s

6.6.1 CRT-RSA

RSA is an asymmetric encryption algorithm. A way to implement this algorithm, yielding faster execution, is known as CRT-RSA. We study this algorithm through three implementations with various protections.

Attacker Model. We consider an attacker able to perform one reset fault in order to conduct a BellCoRe attack [BDL97]. If the output of the CRT-RSA is mathematically linked to the secret key, then, with a number of tries using the algorithm, it is possible to reconstruct the secret key.

The Programs. We base this analysis on the work of Puys et al. [PRBL14]. They describe three versions of CRT-RSA: unprotected, Shamir version and Aumuller version. Only the last one is shown to resist the BellCoRe attack.

Results. We were able to automatically reproduce the attack with 1 reset fault on the unprotected version of CRT-RSA in under 2 seconds. The whole program took just under 3 hours to fully analyze, yielding nine different adversarial paths executing a BellCoRe attack.

Our analysis experienced a timeout (24h) for the remaining two implementations without yielding attack paths. This is not due to a path explosion but to a query complexity explosion. We compare analysis metrics for 1h of analysis and 12 hours. The number of paths explored remained the same, 2 for both, and only x1.8 to x3.5 more instructions were analyzed. However, the average solving time per query went from 0.5 seconds to 1.7 seconds for both, which is a big increase, and significantly higher than the 0.005 seconds in the performance benchmark. No more queries were found trivially true or false by BINSEC/ASE without a solver call between 1h and 12h of analysis.

We summarize those results in Table 6.29.

Table 6.29: Summary of the CRT-RSA security scenario

Version	Ground truth	Result
CRT-RSA basic	Insecure	Insecure ✓
CRT-RSA Shamir	Insecure	Time-out without finding attacks ✗
CRT-RSA Aumuller	Secure	Time-out without finding attacks ✓

Conclusion. We were able to find attacks on the basic CRT-RSA version and found none in Aumuller's version. Our analysis was not able to provide a conclusive result for Shamir's version.

Cryptography is a weak spot of symbolic execution, as it relies on difficult-to-solve problems to avoid reverse-engineering, which is related to what symbolic execution attempts. It is hence not unexpected we would encounter scalability issues for this security scenario. We leave for future work to design optimization techniques to improve our analysis performance of cryptographic programs.

6.6.2 Secret-keeping Machine

Dullien [Dul17] proposes two versions of a toy secret-keeping machine in an exploration of weird states that deviate from normal program behavior and showcases the impact of implementation on program vulnerability.

The Programs. One version is based on linked lists and is manually shown to be exploitable by an attacker able to perform a single bit-flip, while the array version is proven secure against that attacker model. This bit-flip happens in the memory of the secret-keeping machine, modeling a bit-flip from Rowhammer for instance.

The program works as follows. To store a new secret, a user gives the machine a password and a secret. To retrieve their secret, the user sends just their password to the secret-keeping machine, which will output the associated secret.

Attacker Model. Dullien considers an attacker model able to perform one bit-flip in the memory (not in registers or intermediary variables). Its goal is to find the victim's secret without knowing the password.

We start by adapting Dullien's attacker model to BINSEC/ASE, as we do not support the capability 'a bit-flip in memory'. Instead, we use BINSEC/ASE bit-flip capability which happens when a value (register or memory) is updated. Then we manually filter the results to only consider faults in memory loads overwritten afterward and in memory stores.

Attack Scenario. This attack is performed in a scenario of interaction with the secret-keeping machine:

1. The attacker interacts with the machine, storing and retrieving secrets, to set the machine's memory in a specific state;
2. The victim stores a password-secret pair;
3. The attacker interacts again with the machine, induces a bit-flip in memory at some point and enters a value in the machine that should output the victim's secret.

We now present how we experimented with this security scenario.

Experimental Settings. We reimplemented the two versions presented in the paper. One implements the memory with a linked list, the second version uses an array. We simulate the interaction by calling multiple times the secret-keeping machine program with different inputs recreating the scenario proposed in the paper. We discriminate when to inject faults and when not by duplicating the functions interacting with the machine's memory, which is unique and shared. Only the duplicates are faulted.

For this benchmark, we activated faults on variables used as addresses.

Finding a Vulnerability. Our first goal is to reproduce the attack found in the paper

on the vulnerable version. We are happy to report that BINSEC/ASE is able to find the attack described in the paper following the same interaction scenario.

- In memory, data are structured by triplets of password, secret and pointer to the next triplet. When a secret is returned to the user, the program clears the triplet, and puts it on the free linked list, setting the pointer to the next triplet accordingly;
- The attack changes a pointer to the next memory triplet, such that instead of pointing to the next password, it points to its associated secret, both belonging to the attacker;
- Then the attacker sends this secret, acting as a password, to the secret machine, which will output the next pointer and clear the triplet, i.e. the secret, the pointer and the password of the next triplet. The (victim's) password of that next triplet is overwritten by a computable pointer, hence now known to the attacker;
- The attacker sends to the machine the pointer value and the machine outputs the victim's secret.

With this restrictive scenario, it is the only attack we found. It took BINSEC/ASE 0.1s to find it and 7h30min to analyze the whole program.

Program Resistance. Our second goal was to see if we could show that the secure version is indeed secure. We analyze the second, array-built version with the same attacker model. Our analysis was complete and did not report any attacks for a bit-flip in memory.

A More Powerful Attacker Model. Then we explore for which attacker model the resistant version becomes vulnerable. Our analysis of the array-built program shows that by inducing a bit-flip in registers, the attack succeeds. We find the first adversarial path in 78 minutes and analyze the whole program in 13h20min, finding 34 adversarial paths. In the linked-list version, pointers to the next cell are placed in memory and are the target of the attack found. In the array version, array indices provide that function but are stored in registers, hence the array version becomes vulnerable when broadening the specter of the attacker's capability.

Table 6.30: Summary of the Secret keeping machine security scenario

Version	Attacker model	Ground truth	Result
Linked-list	1 bit-flip in memory	Insecure	Insecure ✓
Array	1 bit-flip in memory	Secure	Secure ✓
Array	1 bit-flip anywhere	Insecure	Insecure ✓

Conclusion. To conclude, BINSEC/ASE was able to replay the results from Dullien [Dul17] as summarized in Table 6.30. They highlight the importance of the implementation decisions in program vulnerability. We also show that specifying the attacker model against which to protect a program from is important, as a slight difference can open new attack surface for an attacker.

6.6.3 SecSwift Countermeasure

SecSwift [dF21, CdFB21] is a control-flow integrity (CFI) countermeasure at the llvm level. It is developed by STMicroelectronics.

Experimental Settings. We borrow an implementation of the SecSwift ControlFlow countermeasure from Boespflug *et al.* [BEMP20] at the llvm level. According to the

authors, it associates a unique identifier to each basic block and uses a xor-based mechanism to ensure that the correct branch has been taken. In a nutshell, it prevents the execution from deviating from the CFG. It is a restricted, less powerful version of the real, closed-source, SecSwift countermeasure. We applied the protection to VerifyPIN version 0, the basic version without protections, to study the effects of SecSwift without interference from other protections.

Goal. Our goal is to assess the increased protection provided by the SecSwift countermeasure and to investigate whether we can find attacks on a SecSwift-protected program.

Attacker Model. We consider an attacker able to perform one arbitrary data fault or one test inversion. Their goal is to be authenticated without entering the correct password.

Results. We were able to find nine different attack paths in this binary, all yielding an early loop exit, with either a single test inversion or a single arbitrary data fault. Those attacks take the branch not entering the loop body, or exiting it early, paths that belong to the CFG of the program. These attacks are not unexpected as the adversarial paths stay in the legitimate CFG, yet it is still interesting that our technique finds them automatically. It took BINSEC/ASE 22 minutes to find the first attack and 30 minutes to analyze the whole program. Results are summarized in Table 6.31.

Table 6.31: Summary of the SecSwift security scenario

Version	Ground truth	Result
VerifyPIN_0 with SecSwift	Insecure	Insecure ✓

Conclusion. It is important to characterize exactly against what type of attack a protection is efficient, so they can be combined to provide increased resistance. An incomplete countermeasure may not increase a program's resistance to an attacker.

6.6.4 Neural Network

The CEA List from Grenoble provided us with a very small neural network, based on an implementation from CEA Gardanne [DHM+23]. This neural network is used for evaluating hardware fault injection attack efficiency on microcontrollers for neural networks and highlights their vulnerability. While actual neural networks for embedded system evaluations have tens to hundreds more neurons than this one, it has the same representative architecture (few layers deep, no convolution, ReLU) and allows testing proof of concepts. It has 3 layers of 4,5 and 3 nodes.

The goal of this experiment is to see if an attacker can disturb the computation of the neural network so that it classifies the input incorrectly, finding adversarial examples.

This security scenario was the motivation behind extending BINSEC/ASE support to ARM executables.

Attacker Model. The attacker model chosen is able to perform 1 bit-flip with the goal of making the neural network classify a particular input in the desired incorrect class.

Vulnerabilities Found. Since this program has not been designed to resist fault injection, it is expected that we would find vulnerabilities. BINSEC/ASE found 13

adversarial attack paths in 18 hours, the first attack path was found in 66 seconds. Here are a few examples of successful fault injection:

- Corrupting the load of a weight;
- Corrupting the multiplication between a weight and the value coming from the previous layer;
- Corrupting the formatting function used.

Here, many corruptions result in a wrong computation giving the desired class, but they still need to be targeted depending on the input and the desired class. Table 6.32 summarizes those results.

Table 6.32: Summary of the Neural Network security scenario

Version	Ground truth	Result
Neural Network	Insecure	Insecure ✓

Conclusion. We have shown on this small neural network that an attacker has a wide attack surface at their disposal to corrupt to their wishes the output of the network. This stresses the importance of taking security into account for neural networks, especially for embedded usage and if they perform a security feature.

6.6.5 Security Scenarios Feedback

After those security scenarios, this section provides some feedback on the usability of BINSEC/ASE.

What Was Easy With BINSEC/ASE. There was no real difficulty in configuring BINSEC/ASE once the program was implemented with its oracle. This last part was not always easy though.

What Was Not so Easy With BINSEC/ASE. Understanding BINSEC/ASE output is not trivial for a complex program. In particular, BINSEC/ASE provides the address of activated faults, but only a very loose sense of the attack path, which hinders attack understanding.

Improvments Due to the Security Scenarios. In order to analyze the neural network, which was provided to us as an ARM binary, we had to extend BINSEC/ASE support to this ISA by back-porting it from BINSEC.

6.7 Case Study: WooKey Bootloader

We confront BINSEC/ASE to a real-life security system, WooKey. After presenting WooKey and the goals of this case study, we explore the bootloader, replaying attacks, and evaluate a recent protection scheme, where we found an attack not mentioned before. We propose and evaluate our own patch. Then we explore two other attacks on the WooKey project.

6.7.1 Presentation of WooKey

First presented in 2018 by ANSSI, the French cybersecurity agency, the WooKey platform [BRT+18, Woo] is "a custom STM32-based USB thumb drive with mass storage capabilities designed for user data encryption and protection, with a full-fledged set

of in-depth security defenses". Their choice to be open source and open hardware makes WooKey a relevant case study: it is a real-life, complex device, security-focused and available for reproducibility. Wookey has been extensively analyzed, as it was the target of an ANSSI cybersecurity challenge for security professionals [AAE+20].

We focus on the WooKey bootloader, a dual-bank system enabling hot firmware updates. The system is hardened, especially redundant test protections are present in critical sections to protect against an attacker able to perform one test inversion fault.

6.7.2 Security Scenario and Goal of our Study

We detail in this section the experimental setting and the analysis overview constituting this case study.

WooKey Bootlaoder. We see in Table 6.33 (p.134) that WooKey bootloader size is orders of magnitude larger than the programs used for performance evaluation. Wookey is available as C code. We borrowed C stubs from Lacombe et al. [LFBP21] and we compile the bootloader as we did for the evaluation benchmarks (Section 6.1.2.1).

Attacker Model. We consider the same attack goal as the ANSSI challenge did [AAE+20]: the attacker seeks to manipulate the bootloader logic to boot on the older firmware, more likely to contain security vulnerabilities. We also consider an attacker able to perform a single arbitrary data fault.

Analysis Overview. We conduct the following three analyses:

1. We automatically analyze WooKey at binary-level to check whether we are able to find previously known attacks [LFBP21], and/or new ones in Section 6.7.3: we are indeed able to find the two attacks identified by prior work [LFBP21] (A1, A2), *as well as an attack they did not report* (A3);
2. We automatically analyze at binary level the patched version of Wookey proposed by Lacombe et al. [LFBP21] in Section 6.7.4: we found that the proposed patch indeed blocks the two known attacks (A1 and A2), but not the unreported attack (A3);
3. We propose a definitive patch by adding a counter-measure for A3 and removing parts of the counter-measures that are shown to be useless here (Section 6.7.5). The patch is proven correct w.r.t. our attack model;
4. We explore in Section 6.7.6 other parts of the WooKey project, replaying two other attacks.

6.7.3 Analyze Key Parts of Wookey

The relevant parts of the bootloader's functions are presented in Figure 6.7. Attacks found are summarized in Table 6.34.

Methodology. In this case study, since we only consider an attacker model able to perform one fault (the program is not meant to resist a more powerful attacker), we divide our analysis, faulting only one function at a time. This divide-and-conquer approach allows to obtain fast results.

Results. Lacombe et al. [LFBP21] find an attack in the *loader_exec_req_selectbank* function (A1) and another in the *loader_exec_req_flashlock* function (A2). They correspond to data corruption in branching conditions.

Table 6.33: Benchmarks characteristics and statistics of a standard SE analysis

			BINSEC analysis - no fault			
Program group (#)	C loc	x86 loc	#instructions (explored)	#paths	max #branch in a path	Time
Sections 6.2 to 6.5						
VerifyPINs (8)	80-140	160-215	192-269	1	17-34	< 0.1s
VerifyPIN unrolled (2)	40-85	140-430	142-442	5-17	5-17	< 0.1s
npo2 (2)	50	200-220	607-653	3	31-33	< 0.1s
Section 6.6						
CRT-RSA (3)	125-170	400-600	108k-29M	1	5k-1.3M	0.4s - 1m27
Secret keeping machine (2)	100-200	240-360	1k-1.3k	1	130-150	< 0.1s
VerifyPIN_0 with SecSwift	80	430	430	1	22	< 0.1s
Neural network	100	275	774	1	109	0.1s
Section 6.7						
WooKey bootloader	3.2k	2350	290k	17	18k	9s

```
1  static loader_request_t loader_exec_req_selectbank (loaderstate_t
       nextstate){
2           // ...
3           if((flip_shared_vars.fw.bootable == FW_BOOTABLE &&
                flop_shared_vars.fw.bootable == FW_BOOTABLE) &&
4               !(flip_shared_vars.fw.bootable != FW_BOOTABLE ||
                    flop_shared_vars.fw.bootable != FW_BOOTABLE)){
5                   // ...
6           }
7           if(flop_shared_vars.fw.bootable == FW_BOOTABLE
8   (CM1 - X)    && flip_shared_vars.fw.bootable != FW_BOOTABLE){
9                   if(!(flop_shared_vars.fw.bootable == FW_BOOTABLE
10  (CM2)                && flip_shared_vars.fw.bootable != FW_BOOTABLE))
11                      goto err;
12                  ctx.boot_flop = sectrue ;
13                  // ...
14          }
15          if(flip_shared_vars.fw.bootable == FW_BOOTABLE
16  (CM3 - X)    && flop_shared_vars.fw.bootable != FW_BOOTABLE){
17                  ctx.boot_flip = sectrue ;
18                  // ...
19          }
20          // ...
21  }
22
23  static loader_request_t loader_exec_req_flashlock (loader_state_t
       nextstate){
24          // ...
25          if (ctx.dfu_mode == sectrue) {
26  (CMA)           if (ctx.dfu_mode != sectrue)
27  (CMA)               goto err;
28                  // ...
29          }
30          else if (ctx.dfu mode == secfalse ) {
31                  if (ctx.bootflip == sectrue ) {
32  (CM4)               if (ctx.bootflip != sectrue)
33  (CM4)                   goto err;
34                  // ...
35                  ctx.next_stage = (app_entry_t) (FW1_START);
36                  }
37          // ...
38          }
39  }
```

Figure 6.7: functions of WooKey's bootloader, with [LFBP21] fixes and **our patch**

Table 6.34: Table summarizing the effects of countermeasures

Protection scheme	A1 l.3	A2 l.31	A3 l.25
Normal Wookey	✓	✓	✓
Prior patch (CM1+CM2+CM3+CM4)	✗	✗	✓
Our patch (CM2+CM4+CMA)	✗	✗	✗

Legend - ✓: attack path found by our tool / ✗: no attack found

- For attack A1, faulting the test at line 3 when both firmware flip and flop are bootable results in the execution carrying on to a later test (line 7), which only checks if flop can be booted, assuming at least one firmware cannot. If both are bootable, and flop is the older one, the attacker's goal can be satisfied;
- Attack A2 takes advantage of the lack of countermeasure in *loader_exec_req_flashlock*, which computes the pointer to the boot function of the chosen firmware. Inducing a fault that inverts the test line 31 leads to the wrong pointer being selected.

We are able to find both attacks, linking faults back to their locations in the C code with debug information. *We also find an additional, unreported attack*[7], faulting another part of the *loader_exec_req_flashlock* function (A3).

- In attack (A3), by inverting the test line 25, it is possible to select the wrong firmware pointer to boot on, in particular, if we are not in a secure Direct Firmware Update (DFU) mode, it is possible to boot as if we were.

Remark. While we consider an attacker model able to perform an arbitrary data fault, all attack paths found can equivalently be produced with a test inversion fault, the fault model the WooKey project aims to be resistant to. This shows the scalability of our tool to consider a more powerful attacker model. Interestingly, using arbitrary data faults instead of test inversion does not open new attack paths here.

6.7.4 Analyze a Security Patch of WooKey

We now evaluate the protection scheme proposed by Lacombe et al. [LFBP21] for these attacks. It consists of four extra counter-measures, adding duplicated tests, named from CM1 to CM4, illustrated in Figure 6.7 (p.135).

Our results are summarized in Table 6.34 (p.135). We found indeed that:

- The full protection prevents attacks A1 and A2, as claimed by the authors of the patch;
- Yet, our analysis shows that the protection does not prevent the unreported attack A3.

6.7.5 Propose a New Patch and Evaluate It

Patching Attack A3. We manually inspect the analysis results to understand what happens. We have especially been able to identify the root cause of A3 and propose a dedicated countermeasure for it (named CMA and illustrated in Figure 6.7 (p.135)). It consists again of adding a duplicated test line 26.

Counter-measure Analysis. We analyze each counter-measure in isolation, meaning we implement only one per binary and evaluate each binary separately. We have been able to understand that:

- Counter-measures CM1 and CM3 do not block any attack path as they are redundant with other tests in the code and can be safely removed, for the considered attacker model. As protections often induce an important overhead, only adding useful ones is an interesting area of research. From CM1, we derive the following principle: when there are two nested tests, and no instruction belonging solely to the outer test, doubling the outer test is unnecessary (illustrated Figure 6.8

[7] After discussion with the authors [LFBP21], it turns out that they actually found this path but did not report it in the article, as they did not consider it as a real attack w.r.t. the Wookey challenge.

(p.137)). We have not been able to derive a general rule from the removal of CM3.

– CM2 and CM4 each block one of the attacks found by Lacombe et al.

```
1 if (condition_1) {
2     if (condition_1){
3         if (condition_2) {
4             if (condition_2) {
5                 // code only here
6             }
7         }
8
9     }
10 }
```

Figure 6.8: CM1 double test pattern

Overall, our new patch (CMA + refined former patch) is shown by our tool to protect against all the attacks, for an attacker able to perform one arbitrary data fault, which is an attacker more powerful than the one considered initially in the WooKey project.

6.7.6 Other Attacks on WooKey

We also explore other parts of the WooKey project and we were able to replay two other known attacks.

6.7.6.1 Attack Vectors Combination

The iso8716 library is used in WooKey for secure communication.

Attacker Model. We consider an attacker able to perform one arbitrary data fault. The attack goal (A4) is to induce a buffer overflow to be exploited in a combined attack.

Results. Our analysis found a vulnerability to fault injection which enables a software buffer-overflow in function *SC_get_ATR* [LFBP21]. Contrary to the attack paths found before, this one could not be created with a test inversion fault.

6.7.6.2 Faulty Redundant Test

Martin *et al.* [MKP22] study the correctness of implemented counter-measures. In particular, they were able to detect that one counter-measure doubling a test was incorrectly implemented.

Attacker Model. We consider an attacker able to perform an arbitrary data fault, including on flags. Since we study a small function that directly starts with the double test of interest, this is equivalent to a test inversion fault.

Results. We show that indeed, a redundant test preventing single test inversion faults in the *loader_set_state* function is in fact vulnerable (A5), hence incorrectly implemented as pointed out by Martin et al. We implement the correction they propose and show the correctness of their patch.

6.7.7 Case Study Conclusion

In this case study of WooKey's bootloader, we demonstrated some uses of our tool, in a real-life context. It can replay known attacks, including from source-level analysis and find new ones. It also assists in the countermeasure design and evaluation process. A summary table is presented in Table 6.35.

Table 6.35: Summary of the WooKey bootloader case study

Attacker model		Version	Attack	Ground truth	Result
Capability	Attack goal				
1 AD	boot on older firmware	basic	A1, A2	*Insecure*	*Insecure*
		basic	A3	_	*Insecure*
		CM 1+2+3+4	A1, A2	**Secure**	**Secure**
		CM 1+2+3+4	A3	_	*Insecure*
		CM 2+4	A1, A2	_	**Secure**
		CM 2+4+A	A1, A2	**Secure**	**Secure**
		CM 2+4+A	A3	**Secure**	**Secure**
1 AD	trigger a buffer overflow	basic	A4	*Insecure*	*Insecure*
1 AD	bypass redundant test	basic	A5	*Insecure*	*Insecure*
		basic	A5	**Secure**	**Secure**

AD: arbitrary data fault

Chapter 7

Conclusion

Contents

7.1 Conclusion

Program analysis techniques for safety consider a rather weak attacker model, only able to craft smart inputs. In practice, an advanced attacker has many attack vectors at their disposal, providing them with a wide range of (reusable) capabilities, threatening program security. In this work, we address program analysis in the presence of an advanced attacker. Our goal is to design a bug-finding program analysis technique that is automatic, efficient and generic, integrating a model of an advanced attacker to assess their impact on a program's security properties.

First, we define a model of what we consider to be an advanced attacker. We formalize the impact of this advanced attacker on a program's execution. We extend the standard notion of reachability to adversarial reachability, by including attacker actions as a new type of transition in the transition system representing the program.

Then, we propose a dedicated symbolic algorithm for adversarial reachability, adversarial symbolic execution. We extend symbolic execution by integrating attacker actions as fault injections into the program analysis. A main limitation experienced by the state-of-the-art techniques is path explosion, we mitigate it by integrating a novel forkless encoding of attacker actions. Furthermore, we design two optimizations dedicated to reducing the number of fault injections in queries to alleviate the solver's task. Early Detection of fault Saturation aims to stop fault injection as soon as possible, while Injection On Demand only adds faults when necessary to explore a branch, injecting them as late as possible. Both preserve the attacker model's power.

We implement adversarial symbolic execution on top of the BINSEC framework for binary-level analysis. While adversarial symbolic execution is independent of the analysis abstraction level, what is executed on the processor, and hence faulted, is the binary program, motivating our implementation abstraction level. Our technique is shown to significantly reduce the number of paths to explore, and scales up to 10 faults on a standard SWIFI benchmark, where prior forking attempts timeout for 3 faults.

We illustrate the interest and feasibility of our technique through different security scenarios, replaying attacks of attackers with various capabilities. Also, we show that our method scales to realistic size examples, such as the WooKey project where we have been able to replay known fault attacks and even find a vulnerability not reported in a recently proposed countermeasure patch.

Essentially, this work is a first step in designing efficient program analysis techniques able to take into account advanced attackers.

7.2 Perspectives

We present several directions to extend or improve the work presented in this thesis.

Extend the Supported Attacker Model. While our formalization allows for a generic attacker model, our implementation of adversarial symbolic execution is more restricted in attacker capabilities. For instance, it would be interesting and would widen the scope of this tool's usability to implement an attacker model able to perform actions from different fault models in one attack. Similarly, our technique would benefit from having an efficient algorithm to deal with fault on addresses, for instance by recording seen addresses deemed 'interesting', or having a preprocessing selecting a set of 'interesting' addresses. It would allow corrupting pointers, for instance, to point to the reference password instead of the user input password in the password comparison of an authentication program. Efficient faults on addresses can also be used for advanced control-flow attacks, such as return-oriented programming (ROP), when reasoning about code addresses. Currently, we support those faults with a combinatorial explosion. We are also missing an efficient algorithm for general instruction corruption. Another angle would be to extend the supported fault models to broaden the scope of represented attacks. In particular, micro-architectural components could be modeled to account for more attacks targeting micro-architectural elements such as physical fault injection on the prefetch buffer.

Such extensions would likely increase the complexity of the analysis. We believe more optimizations can be designed to minimize the burden of fault injection on symbolic execution, working towards reducing query complexity. It would also be interesting to see how different fault encodings are in fact handled inside the solver to select or design the best one. It could even be a mix of encoding.

A Hybrid Forking/Forkless Technique. We have seen in Chapter 6 that not all programs and not all fault models respond best to a forkless fault encoding. At first glance, forkless encodings seem best suited for data faults while forking encodings could be favored for control-flow faults. It would be interesting to characterize what types of programs respond best to a forkless technique and which respond best to a forking one. Going further, we believe that inside a program, some possible fault locations could respond best to one technique or the other. Hence, a future research direction would be to design a hybrid forkless/forking technique and heuristics to choose which technique to use on which possible fault location. Interestingly, this problem relates to path merging in symbolic execution, where heuristic split cases into either fork the analysis (analog to forking fault injection) or keep a merged state (analog to forkless fault injection). The best performance for the path merging problem is found with a hybrid technique, motivating this future research direction.

Finding the Minimal Attacker. The approach presented in this thesis focuses on as-

sessing the vulnerability or resistance of a program to an attacker model. The problem can also be seen from the opposite perspective, that is to say, having an analysis find a 'minimal' attacker model for which a program is vulnerable. Here, an order that makes sense between various fault models and number of faults considered would have to be coined.

Appendix A

Éléments de traduction en français

French summary of the thesis. The remaining of this appendix will be written in French.

Une introduction substantielle est proposée Section A.1, suivie d'un résumé de chaque chapitre Section A.2, terminant par une conclusion Section A.3.

A.1 Chapitre 1 : Introduction

Contexte. Des programmes se trouvent partout dans nos vies : des smartphones dans nos poches aux ordinateurs sur lesquels nous travaillons, nos télévisions, les systèmes informatiques du supermarché du coin, mais aussi les puces de nos cartes de crédits, nos systèmes de transport, de nos hôpitaux, les systèmes de distribution d'eau, d'électricité, etc. Ces programmes peuvent contenir des fonctionnalités de sécurité, dont le mauvais fonctionnement peut avoir de graves conséquences en termes monétaires, matérielles, humaines, ou en termes de fuite de secret.

Cependant, les programmes étant écrits à la main par des développeurs, des erreurs s'y immiscent. Ces bogues peuvent ensuite être exploités par des attaquants, par exemple, utiliser un dépassement de tampon pour lire ou écrire des données, ou exécuter du code arbitraire. Des protections existent mais restent souvent incomplètes et le programme continue à être vulnérable face à un attaquant avancé, par ailleurs capable d'utiliser des vecteurs d'attaque autres que les vulnérabilités logicielles tel que l'injection de faute durant l'exécution du programme. De nouveaux vecteurs et techniques d'attaque sont régulièrement découverts.

De nombreux travaux de recherche, ces dernières décennies, se sont intéressés au problème de *l'analyse automatique de programme* afin de les rendre plus robustes. De nombreuses techniques ont été développées telles que l'exécution symbolique [CS13, GLM12, BGM13], l'analyse statique [Fac], l'interprétation abstraite [CGJ+03] ou l'analyse bornée de modèles [CBRZ01], détectant des bogues ou cherchant à montrer leur absence [CCF+05, KKP+15], conduisant à une adoption industrielle [BGM13, Fac, BCLR04, KKP+15, LRV+22]. S'assurer qu'un programme fonctionne comme voulu, c'est-à-dire sans bogue, relève de la *sûreté* de fonctionnement. Comme les bogues sont une porte d'entrée aux attaques, les supprimer est un premier pas vers la *sécurité* logicielle.

Problème. Ces analyses de programmes pour la sûreté se reposent sur un modèle d'attaquant faible, seulement capable de concevoir des entrées malveillantes. En pra-

tique, un attaquant avancé est capable d'exploiter des vulnérabilités complexes pour monter des attaques micro-architecturales ou encore des attaques matérielles, physiques ou contrôlées par logiciel tel que Rowhammer. Il peut également combiner ces différents vecteurs d'attaque.

Pour construire une analyse de programme pour la sécurité, grâce à laquelle il serait possible de rejouer des attaques, faisant varier différents paramètres et d'évaluer des protections logicielles, nous avons identifié le besoin de réconcilier analyse de programme et modèle d'attaquant avancé. Cependant, prendre en compte les nombreuses actions possibles d'un attaquant augment significativement l'espace d'état du programme à analyser, menaçant la capacité de l'analyse à passer à l'échelle.

Objectif et Défis. *Notre objectif est de concevoir une technique automatique, efficace et générique pour raisonner sur l'impact d'un attaquant avancé sur les propriétés de sécurité d'un programme* alors que les analyses standards ne considèrent qu'un attaquant capable de créer des entrées malicieuses.

Notre premier défi consiste à modéliser un attaquant avancé et à imaginer une formalisation de son impact sur l'exécution d'un programme. Notre second défi est de concevoir un algorithme efficace et générique pour étudier la vulnérabilité d'un programme à un modèle d'attaquant donné, tout en maîtrisant la complexité inhérente à l'ajout d'actions possible pour l'attaquant, en particulier lorsque plusieurs actions sont considérées durant une seule attaque.

Les rares techniques existantes dans le domaine se concentrent sur les injections de fautes matérielles pour des composants de sécurité. Elles se basent sur des techniques comme la *génération de mutants* [CCG13, RG14, GWJLL17, CDFG18, GWJL20] ou des *analyses branchantes* [PMPD14, BBC+14, BHE+19, LFBP21, Lan22]. Ces techniques souffrent de difficultés de passage à l'échelle, en particulier pour considérer plusieurs fautes lors d'une attaque. De plus, elles sont généralement limitées à quelques modèles de fautes pré-définis.

Proposition. Nous proposons un modèle caractérisant un attaquant avancé en fonction de ses capacités à modifier le comportement du programme et de son objectif d'attaque. Afin de pouvoir raisonner sur ce modèle d'attaquant, nous présentons une extension du système de transitions représentant l'exécution d'un programme avec de nouvelles transitions modélisant les capacités de l'attaquant. *L'atteignabilité adversariale* est un formalisme exprimant l'atteignabilité de l'objectif de l'attaquant dans ce système de transitions étendu.

Nous construisons un nouvel algorithme basé sur l'exécution symbolique, appelé *exécution symbolique adversariale*, pour répondre au problème de l'atteignabilité adversariale selon l'angle de la détection de bogue (vérification bornée). Notre algorithme est générique en termes de support de capacités attaquant et permet d'atténuer l'explosion du nombre de chemins à analyser grâce à un encodage non-branchant des capacités de l'attaquant, présentées comme des fautes injectées. Afin d'améliorer davantage les performances, nous proposons deux optimisations permettant de réduire le nombre de fautes injectées. Nous montrons enfin que notre algorithme est correct et k-complet pour l'atteignabilité adversariale.

Nous implémentons cet algorithme pour l'analyse de programme binaire dans la plateforme BINSEC d'exécution symbolique, construisant BINSEC/ASE.

Contributions. Dans cette thèse, nous revendiquons les contributions suivantes :

- Nous proposons un modèle représentant un attaquant avancé et nous formalisons son impact sur les propriétés d'un programme avec *l'atteignabilité adversariale*,

une extension du concept d'atteignabilité en présence d'un attaquant avancé. Nous définissons les propriétés de correction et de complétude associées ;
- Nous décrivons *l'exécution symbolique adversariale*, une nouvelle technique symbolique évaluant l'atteignabilité adversariale. L'objectif de cet algorithme est d'être automatique et efficace dans son raisonnement sur l'impact d'un attaquant avancé sur un programme, et d'être générique quant au support du modèle d'attaquant. Notre algorithme comprend un nouvel encodage non-branchant des capacités de l'attaquant pour atténuer l'explosion de chemins et deux optimisations pour limiter le nombre d'injections tout en préservant le même modèle d'attaquant. Nous montrons que notre algorithme est correct et k-complet pour l'atteignabilité adversariale ;
- Nous proposons une implémentation de l'exécution symbolique adversariale pour l'analyse de programme binaire au sein de la plateforme BINSEC, construisant un outil appelé *BINSEC/ASE*. Nous procédons à une évaluation expérimentale de la performance de BINSEC/ASE comparé à l'état de l'art en utilisant des programmes de test issus du domaine de l'injection de fautes matérielles et des cartes à puce. Nous montrons un gain significatif en termes de temps d'analyse et de chemins explorés. Nous soulignons l'intérêt et la faisabilité de notre technique en explorant plusieurs scénarios de sécurité. Enfin, nous étudions le cas du programme de démarrage de WooKey, accomplissant une étude de vulnérabilité de deux implémentations et proposons un correctif pour une attaque non rapportée précédemment.

Ces travaux sont un premier pas vers la conception d'analyse de programme efficace pour la sécurité, prenant en compte un attaquant avancé. Notre approche est suffisamment générique pour inclure de nombreuses capacités attaquant comme l'inversion de bit de Rowhammer, l'inversion de test, le saut d'instruction ou la corruption arbitraire de données. Cependant, la modification d'instructions reste hors de notre portée. Alors que nous avons investigué le côté détection de vulnérabilité (sous-approximation) de l'atteignabilité adversariale, le coté vérification (sur-approximation) serait intéressant à explorer également. Ce sont des directions passionnantes pour de futurs travaux de recherche.

Une partie des travaux présentés dans cette thèse ont été publiés à ESOP 2023 [DBP23].

A.2 Résumés de chaque chapitre

A.2.1 Chapitre 2 : Contexte et Motivation

Ce chapitre décrit le contexte des systèmes d'information implémentant des fonctionnalités de sécurité, bien plus répandus que l'on peut le penser, ainsi que les vulnérabilités auxquelles ils font face et les processus de sécurisation mis en place. Nous motivons le besoin d'intégrer un modèle d'attaquant avancé dans les analyses de programme en listant quelques vecteurs d'attaque à la disposition d'un tel attaquant, notamment l'injection de faute, et les capacités obtenues.

A.2.2 Chapitre 3 : Préambule

Dans ce chapitre, nous discutons de quelques notions importantes pour la compréhension de la suite du manuscrit. L'analyse de programme, les différents type d'approximations et quelques propriétés qui peuvent être vérifiées dans un programme sont présentés. En particulier, la notion d'atteignabilité y est définie. Nous présentons une technique d'analyse de programme, l'exécution symbolique, utilisée dans cette thèse. Nous abordons ensuite les particularités de l'analyse de programme au niveau binaire, et de l'analyse de programme pour l'injection de faute.

A.2.3 Chapitre 4 : Atteignabilité Adversariale

Ce chapitre débute par la formalisation du modèle d'attaquant utilisé, composé (1) des capacités d'attaque, (2) d'un nombre maximal d'actions et (3) d'un objectif d'attaque. Nous intégrons ce modèle d'attaquant dans la notion d'atteignabilité, définissant ainsi la notion d'atteignabilité adversariale, en étendant la représentation d'un programme sous forme d'un système de transition avec de nouvelles transitions adversariales. La correction et complétude d'une analyse pour l'atteignabilité adversariale sont définies. Un algorithme de vérification de l'atteignabilité adversariale est ensuite proposée, l'exécution symbolique adversariale, basée sur l'exécution symbolique. Les actions de l'attaquant sont modélisées par des fautes, encodées de façon non branchante dans la construction du prédicat de chemin. Deux optimisations permettant de réduire le nombre de fautes dans les requêtes au solveur sont décrites.

A.2.4 Chapitre 5 : le Prototype BINSEC/ASE

Dans ce chapitre, nous décrivons l'implémentation de notre algorithme d'exécution symbolique adversariale dans l'outil BINSEC, pour l'analyse de code au niveau binaire. En particulier, nous changeons la manière dont sont traitées les instructions pour y inclure la possibilité d'une faute. Nous détaillons également l'implémentation de la technique branchant dans BINSEC, qui construit le compétiteur face auquel nous comparer. Enfin, nous proposons un guide utilisateur pour utiliser BINSEC/ASE avec un nouveau programme, et un guide développeur pour ajouter de nouveaux modèles de fautes.

A.2.5 Chapitre 6 : Evaluation Expérimentale

Ce chapitre contient l'évaluation expérimentale de notre outil BINSEC/ASE. En particulier, nous montrons que notre technique d'injection améliore significativement les performances de l'analyse, d'autant plus lorsque le nombre de fautes considéré est élevé. Notre technique permet d'atténuer l'explosion du nombre de chemins à laquelle l'état de l'art fait face.

Nous montrons également l'intérêt et la faisabilité de notre technique en explorant différents scénarios de sécurité : reproduire une attaque BellCoRe sur CRT-RSA, étudier la différence de vulnérabilité de deux implémentations d'une boîte à secret, chercher des attaques sur la contre-mesure SecSwift, étudier l'impact d'un attaquant sur un réseau de neurones, et enfin, analyser le cas du programme de démarrage de WooKey, un challenge de sécurité de l'ANSSI. Nous sommes capables de rejouer des attaques et avons trouvé une attaque non rapportée dans un correctif récent.

A.3 Chapitre 7 : Conclusion

Les techniques d'analyse de programme se concentrent majoritairement sur l'aspect sûreté et considèrent un modèle d'attaquant faible, seulement capable de créer des entrées malicieuses pour les programmes. En pratique, un attaquant puissant peut tirer parti de nombreux vecteurs d'attaques, menaçant la sécurité des programmes. Dans cette thèse, nous visons à porter l'analyse de programme de la sûreté à la sécurité par la modélisation et l'intégration d'un modèle d'attaquant avancé et de son impact sur les propriétés de sécurité d'un programme.

Nous avons d'abord défini un modèle d'attaquant représentant un attaquant avancé et nous avons formalisé son impact sur l'exécution d'un programme au travers de l'atteignabilité adversariale qui étend le système de transition représentant un programme avec des transitions adversariales modélisant les actions de l'attaquant.

Nous avons ensuite conçu un algorithme, l'exécution symbolique adversariale pour répondre au problème de l'atteignabilité adversariale. Les techniques de l'état de l'art souffrent d'une explosion du nombre de chemins à explorer. Nous atténuons ce phénomène par un encodage non branchant des capacités de l'attaquant et deux optimisations permettant de réduire la complexité des requêtes au solveur.

Puis, nous avons implémenté cet algorithme dans le moteur d'exécution symbolique BINSEC, créant une analyse de sécurité au niveau binaire. Notre évaluation expérimentale montre une amélioration significative des performances de l'analyse par rapport à la technique branchante de l'état de l'art, en parallèle d'une réduction manifeste du nombre de chemins à explorer.

Enfin, nous illustrons l'intérêt et la faisabilité de notre technique au travers de différents scénarios de sécurité. Nous étudions également le cas du programme de démarrage de WooKey, un challenge de sécurité de l'ANSSI. Nous sommes capables de trouver des attaques connues, et avons même mis en évidence une attaque non rapportée précédemment sur un correctif récent et proposé notre propre correctif.

Travaux Futurs. Nous présentons maintenant quelques perspectives de travaux futurs étendant ou améliorant les travaux présentés dans cette thèse :

- Étendre le support de BINSEC/ASE à de nouveaux modèles de faute, avec un besoin d'algorithmes efficaces. Par exemple, notre formalisation est générique et permet de modéliser un attaquant avec plusieurs types d'actions différentes, ce qui n'est, à l'heure actuelle, pas implémenté dans notre outil. Il serait également intéressant de poursuivre la recherche d'optimisations limitant en amont le nombre de points d'injection possibles ;
- Certains programmes, modèles de fautes, ou peut-être même points d'injection se révèlent plus long à traiter avec une analyse non branchante. À première vue, une analyse non branchante semble être plus performante sur des fautes sur les données, alors qu'une analyse branchante pourrait avoir de meilleur résultat pour des fautes sur le flow de control. Il serait intéressant de chercher des heuristiques pour déterminer quand utiliser l'une ou l'autre des techniques, créant une analyse hybride ;
- Dans cette thèse, nous nous intéressons à montrer la vulnérabilité ou la résistance d'un programme à un modèle d'attaquant donné. À l'inverse, on pourrait chercher quel est l'attaquant "minimal" pouvant attaquer un programme. Il serait alors nécessaire de proposer une relation d'ordre entre les modèles de faute et leurs nombres qui fasse sens.

Appendix B

Additional Experimental Data

In this appendix chapter, we include additional experimental data that did not fit in Chapter 6. The interested reader will find:

- A deeper evaluation of the reset fault model in Section B.1 (RQ B1), assessing the impact of our optimizations (RQ B1.1) in Section B.1.1 and the impact of the forkless encoding (RQ B1.2) in Section B.1.2;
- An evaluation of the impact of our optimizations on bit-flip faults (RQ B2) in Section B.2;
- An evaluation of the impact of our optimizations on instruction skip faults (RQ B3) in Section B.3;
- A summary of our optimization evaluation across fault models in Section B.4;
- An evaluation of the affinity of different solvers to our various arbitrary data forkless encodings (RQ B4) in Section B.5.

B.1 FASE Evaluation of Reset Faults (RQ B1)

We evaluate BINSEC/ASE for the reset fault model regarding the impact of our optimizations and the impact of the chosen forkless encoding. We assess them independently. Our aim is to verify the extensibility of arbitrary data fault results.

B.1.1 Impact of Optimizations (RQ B1.1)

We evaluate now the impact of the different optimizations, EDS, IOD and their combination, IOD+EDS, for reset faults.

RQ B1.1. What is the impact of our optimizations on reset faults?

Goal. The goal of this experiment is to evaluate our optimizations for the reset fault model and evaluate how they contribute to reducing query complexity. In particular, we want to check whether general trends for arbitrary data fault evaluation also hold for reset faults.

Protocol. We evaluate FASE on the performance benchmark, with an attacker model able to perform 1 to 10 reset faults.

Results. We start by checking which is the fastest optimization. Analysis time results are presented in Table B.1 (p.VIII). On average for all numbers of faults, FASE-EDS is x1.05 times faster than FASE, FASE-IOD x1.6 times faster and FASE-EDS+IOD x1.4 times faster.

Table B.1: Analysis time for reset faults (RQ B1.1)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
average value							
FASE-Reset	0.815	2.32	4.53	7.48	12.4	17.5	19.6
FASE-Reset-EDS	0.561	1.85	4.42	7.56	13.9	20.0	24.0
FASE-Reset-IOD	0.363	1.15	2.81	5.04	9.78	13.3	15.2
FASE-Reset-EDS+IOD	0.414	1.3	3.15	5.65	10.8	14.9	16.9
Forking-Reset	0.922	18.8	255	2.82k	3.69k	27.8k	11.3k
timeouts (24h) over 12 benchmarks in total							
FASE-Reset	0	0	0	0	0	0	0
FASE-Reset-EDS	0	0	0	0	0	0	0
FASE-Reset-IOD	0	0	0	0	0	0	0
FASE-Reset-EDS+IOD	0	0	0	0	0	0	0
Forking-Reset*	0	0	0	0	9	9	11

(Forking-Reset*: values for 6 faults and above are computed for incomplete runs due to timeouts)

To measure the impact of our optimizations on query complexity, we consider the ratio of queries created on the number of queries sent to the solver, presented in Table B.2 (p.IX). This provides information about the complexity of the created queries and how many can be solved to true or false with arithmetic rules inside BINSEC without a solver call. We can see a trend similar to arbitrary data faults, where optimizations increase the number of queries created by the analysis, but allow for simpler queries, as many more are simplified without the solver's help.

Finally, our second measure of query complexity is the average solving time per query, assessing the complexity of queries for those that need to be sent to the solver. Results for average solving time per query are presented in Table B.4 (p.X). The average solving time per query also decreases with optimizations. However, while FASE-IOD managed to get to the level of the forking technique for arbitrary data faults, here the forking technique remains x2.3 times faster than FASE-IOD on average for all numbers of faults.

Conclusion RQ B1.1. Reset faults experimentally show trends in performance similar to arbitrary data faults, with FASE-IOD still the best optimization.

Table B.2: Number of queries created and sent to the solver for reset faults (RQ B1.1)

	Number of queries created / Number of queries sent to the solver						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE-Reset	185 / 80.8	465 / 204	899 / 383	1.38k / 583	2.16k / 939	2.64k / 1.21k	2.9k / 1.36k
FASE-Reset-EDS	198 / 65.6	518 / 211	1.04k / 441	1.65k / 702	2.66k / 1.19k	3.3k / 1.56k	3.66k / 1.8k
FASE-Reset-IOD	198 / 48.2	527 / 176	1.08k / 410	1.76k / 712	2.9k / 1.29k	3.59k / 1.71k	3.96k / 1.95k
FASE-Reset-EDS+IOD	205 / 53.0	555 / 191	1.15k / 444	1.87k / 765	3.11k / 1.38k	3.87k / 1.85k	4.32k / 2.15k
Forking-Reset*	1.34k / 173	29.2k / 4.08k	476k / 69.0k	6.22M / 963k	8.27M / 2.04M	61.2M / 20.8M	13.1M / 6.0M

(Forking-Reset*: values for 6 faults and above are computed for incomplete runs due to timeouts)

Table B.3: Number of queries created and sent to the solver comparison for reset fault encodings (RQ B1.2)

	Number of queries created / Number of queries sent to the solver						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE-Reset-Ite	185 / 80.8	465 / 204	899 / 383	1.38k / 583	2.16k / 939	2.64k / 1.21k	2.9k / 1.36k
FASE-Reset-Sub	185 / 103	465 / 271	899 / 521	1.38k / 801	2.16k / 1.26k	2.64k / 1.59k	2.9k / 1.79k
FASE-Reset-And	185 / 103	465 / 271	899 / 521	1.38k / 801	2.16k / 1.26k	2.64k / 1.59k	2.9k / 1.79k
FASE-Reset-Xor	185 / 103	465 / 271	899 / 521	1.38k / 801	2.16k / 1.26k	2.64k / 1.59k	2.9k / 1.79k

Table B.4: Average solving time per query for reset faults (RQ B1.1)

	Average solving time per query (s)						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE-Reset	0.007	0.009	0.009	0.009	0.010	0.010	0.009
FASE-Reset-EDS	0.007	0.008	0.009	0.009	0.009	0.009	0.009
FASE-Reset-IOD	0.005	0.006	0.007	0.006	0.006	0.007	0.006
FASE-Reset-EDS+IOD	0.005	0.007	0.007	0.006	0.007	0.007	0.006
Forking-Reset*	0.003	0.003	0.003	0.003	0.001	0.001	0.002

(Forking-Reset*: values for 6 faults and above are computed for incomplete runs due to timeouts)

B.1.2 Comparison of the Different Forkless Encodings (RQ B1.2)

Multiple forkless encodings have been implemented for reset faults (see Section 5.3.3.1). We compare them here.

RQ B1.2. What is the impact of the different encodings on the solver's performance?

Goal. Our goal is to explore the impact of different operators in the forkless encoding and their affinity with the solver used by BINSEC/ASE. We check whether the ite operator also makes for the fastest forkless encoding with reset faults.

Protocol. We evaluate FASE on the performance benchmark, with an attacker model able to perform 1 to 10 reset faults. We compare four encodings, FASE-Reset-Ite, FASE-Reset-Sub (with a munis operator), FASE-Reset-And (with a logical and operator) and FASE-Reset-Xor (with a logical xor operator), on three metrics. No optimizations were used.

Results. First, analysis time results are presented in Table B.5 (p.X). In terms of overall analysis time, FASE-Reset-Ite is x2.1 times faster than FASE-Reset-Xor for 1 fault on average, x3.5 times faster than FASE-Reset-And and x3.4 times faster than FASE-Reset-Sub.

Table B.5: Analysis time for reset faults (RQ B1.2)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE-Reset-Ite	0.815	2.32	4.53	7.48	12.4	17.5	19.6
FASE-Reset-Sub	2.79	7.03	14.0	23.5	39.4	52.0	55.2
FASE-Reset-And	2.86	7.78	15.8	25.9	45.7	59.7	71.6
FASE-Reset-Xor	1.72	4.87	10.0	16.2	26.9	34.2	39.3

Then, looking at query complexity, our first metric is the ratio of queries created on the number of queries sent, presented in Table B.3 (p.IX), reflecting query complexity at query creation inside BINSEC, before solver call. FASE-Reset-Ite enables more arithmetical simplifications preventing a solver call for all fault numbers. FASE-Reset-Ite simplifies 56% of queries, while the other 3 encodings only simplify 44% of queries. This difference is likely due to the various arithmetic simplification rules implemented

in BINSEC, not treating concrete constant values (in reset faults) like more general expressions (in arbitrary data faults).

Finally, the average solving time per query results are presented in Table B.6 (p.XI), to reflect the query complexity in solver calls. Due to variability in the measurements, 100 runs were averaged for each program of the benchmark. FASE-Reset-Ite is also the encoding generating queries that take the least time to solve. Averaging over all number of faults, FASE-Reset-Ite is x3.4 times faster than FASE-RESET-Sub, x3.7 times faster than FASE-Reset-And and x1.8 times faster than FASE-Reset-Xor.

Table B.6: Average solving time per query for reset faults (RQ B1.2)

	Average solving time per query (s)						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE-Reset-Ite	0.007	0.009	0.009	0.009	0.010	0.010	0.009
FASE-Reset-Sub	0.031	0.030	0.029	0.035	0.028	0.028	0.029
FASE-Reset-And	0.032	0.036	0.035	0.030	0.031	0.033	0.035
FASE-Reset-Xor	0.014	0.016	0.017	0.017	0.017	0.017	0.017

Conclusion RQ B1.2. The encoding using the ite operator is the best-performing encoding for reset faults and arbitrary data faults for our BINSEC/ASE implementation and the solver it uses.

B.2 FASE Evaluation of Bit-Flip Faults (RQ B2)

In this section, we assess the last data fault model implemented by evaluating our different optimizations (FASE+EDS, FASE+IOD and FASE-EDS+IOD) and FASE. We have only one forkless encoding implemented, using an ite and a shift operator, for the bit-flip fault model.

RQ B2. What is the impact of our optimizations on bit-flip faults?

Goal. The goal of this experiment is to check that FASE-IOD also performs best for bit-flips and evaluate how optimizations contribute to reducing query complexity.

Protocol. We evaluate FASE on the performance benchmark, with an attacker model able to perform 1 to 10 bit-flip faults.

Results. We present the analysis time results in Table B.7 (p.XII). FASE IOD is x2.2 times faster than FASE, x1.4 times faster than FASE-EDS and x1.1 times faster than FASE-EDS+IOD.

To compare query complexity, we start with the ratio of queries created over queries sent to the solver, presented in Table B.9 (p.XIII). As expected, optimizations generate some more queries than FASE, x1.6 times for FASE-EDS and FASE-IOD for 1 fault, and x1.1 times more for FASE-EDS+IOD. Despite this, they succeed in reducing query complexity as more queries are arithmetically resolved to true or false by BINSEC without the help of the solver. FASE-EDS sends x1.1 times fewer queries to the solver, FASE-IOD x1.4 times fewer and FASE-EDS+IOD x1.2 times fewer. As the number of faults increases, the ratio of queries simplified without a solver call increases also.

Table B.7: Analysis time for bit-flip faults (RQ3.2)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
average value							
FASE	2.71	5.42	7.97	9.84	13.0	13.8	14.2
FASE-EDS	1.71	4.58	7.87	10.3	15.1	15.2	16.0
FASE-IOD	1.25	2.89	4.69	6.35	8.76	9.27	9.15
FASE-EDS+IOD	1.4	3.82	5.25	6.9	9.51	10.4	10.5
Forking*	0.645	15.1	225	952	2.13k	13.4k	9.47k
timeouts (24h) over 12 benchmarks in total							
FASE	0	0	0	0	0	0	0
FASE-EDS	0	0	0	0	0	0	0
FASE-IOD	0	0	0	0	0	0	0
FASE-EDS+IOD	0	0	0	0	0	0	0
Forking	0	0	0	2	7	8	10

(Forking*: values for 4 faults and above are computed for incomplete runs due to timeouts)

Our second measure of query complexity is the average solving time per query, presented in Table B.8 (p.XII). On average over all faults, FASE-EDS queries are x1.04 faster to solve, FASE-IOD x1.9 times and FASE-EDS+IOD x1.8 times faster.

Table B.8: Average solving time per query for bit-flip faults (RQ B2)

	Average solving time per query (s)						
	1f	2f	3f	4f	6f	8f	10f
average value							
FASE	0.050	0.053	0.052	0.046	0.050	0.050	0.050
FASE-EDS	0.035	0.045	0.060	0.049	0.057	0.048	0.053
FASE-IOD	0.035	0.024	0.024	0.024	0.026	0.026	0.027
FASE-EDS+IOD	0.034	0.037	0.024	0.024	0.025	0.026	0.027
Forking*	0.003	0.004	0.003	0.002	0.003	0.004	0.004

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts)

Conclusion RQ B2. FASE-IOD is still the fastest optimization for the bit-flip fault model.

Table B.9: Number of queries created and sent to the solver for bit-flip faults (RQ B2)

	Numbner of queries created / Number of queries sent to the solver						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE	161 / 86.1	303 / 162	435 / 228	547 / 287	676 / 358	715 / 377	726 / 382
FASE-EDS	171 / 78.0	330 / 165	480 / 243	609 / 313	766 / 407	813 / 432	826 / 439
FASE-IOD	171 / 60.8	345 / 144	517 / 222	660 / 292	823 / 370	870 / 388	883 / 393
FASE-EDS+IOD	181 / 70.8	372 / 166	557 / 250	712 / 321	902 / 416	960 / 440	976 / 447
Forking*	1.06k / 237	17.8k / 5.24k	204k / 68.0k	1.05M / 349k	2.09M / 703k	8.18M / 3.07M	5.19M / 2.15M

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts)

Table B.10: Number of queries created and sent to the solver comparison for instruction skip faults (RQ B3)

	Number of queries created / Number of queries sent to the solver						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE*	274 / 156	951 / 535	2.52k / 1.32k	5.41k / 2.76k	14.0k / 7.29k	20.8k / 10.5k	25.9k / 12.7k
FASE-IOD*	352 / 114	1.25k / 519	3.3k / 1.4k	6.99k / 2.94k	17.3k / 7.44k	24.5k / 10.1k	30.5k / 12.4k
Forking*	2.8k / 474	133k / 34.0k	2.76M / 496k	69.0M / 4.67M	46.7M / 13.6M	181M / 32.3M	133M / 33.8M

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts
FASE*: all value from incomplete runs)

B.3 FASE Evaluation of Instruction Skip Faults (RQ B3)

We now evaluate BINSEC/ASE's performance for instruction skips. Only one forkless encoding has been implemented, using an ite operator. FASE support instruction skips in its un-optimized version, and with IOD.

RQ B3. What is the impact of our optimizations on instruction skip faults?

Goal. The goal of this experiment is to check whether IOD improves FASE's performance and evaluate how it contributes to reducing query complexity.

Protocol. We evaluate FASE on the performance benchmark, with an attacker model able to perform 1 to 10 instruction skip faults.

Results. We start with the analysis time results, presented in Table B.11 (p.XIV). First, we can see that there is 1 program for which the instruction skip forkless cannot finish in time for FASE and FASE-IOD similarly. It is the unrolled version of VerifyPIN with a PIN size of 16. This is a rather long sequence of assignments that seem to overload the analysis when faulted in a forkless manner. As FASE and FASE-IOD timeout for the same benchmark, we can still compare their results in analysis time. We see that FASE-IOD is x1.1 times faster than FASE on average for 1 fault, which is less than for data fault models. Injection On Demand was designed with data faults in mind, we leave to future work to imagine an optimization tailored for this particular fault model.

Table B.11: Analysis time for instruction skip faults (RQ B3)

	Analysis time (s)						
	1f	2f	3f	4f	6f	8f	10f
average value							
FASE*	107	175	343	625	1.73k	2.94k	3.23k
FASE-IOD*	95.6	150	329	515	822	842	962
Forking*	9.6	857	3.52k	16.3k	32.1k	86.4k	86.4k
timeouts (24h) over 12 benchmarks in total							
FASE	1	1	1	1	1	1	1
FASE-IOD	1	1	1	1	1	1	1
Forking	0	0	3	6	10	12	12

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts
FASE*: all value from incomplete runs)

Then, we evaluate FASE-IOD compared to FASE on query complexity metrics. The ratio of queries created over queries sent is presented in Table B.10 (p.XIII). As runs are incomplete for at least one benchmark, comparison of exact values does not make sense, but trends are preserved. Hence we can see that, as expected, FASE-IOD creates more queries but reduces query complexity enough so that it sends fewer to the solver than FASE.

Our second query complexity metric, the average solving time per query, is presented in Table B.12 (p.XV). As for other fault models, FASE-IOD allows to greatly reduce the average solving time per query. On average over all number of faults, FASE-IOD has queries x5.4 faster to solve than FASE.

Table B.12: Average solving time per query for instruction skip faults (RQ B3)

	Average solving time per query (s)						
	1f	2f	3f	4f	6f	8f	10f
	average value						
FASE*	0.051	0.085	0.138	0.143	0.135	0.132	0.140
FASE-IOD*	0.017	0.025	0.023	0.020	0.023	0.021	0.023
Forking*	0.010	0.012	0.006	0.005	0.002	0.003	0.003

(Forking*: values for 3 faults and above are computed for incomplete runs due to timeouts
FASE*: all value from incomplete runs)

Conclusion RQ B3. FASE-IOD also reduces query complexity for the instruction skip fault model, which allows for a faster analysis.

B.4 FASE Optimizations Summary

FASE-IOD is our best-performing optimization, and thus across all fault models for which it is implemented. This gives us hope that it will be efficient too for yet another fault model useful to a BINSEC/ASE user. It is interesting to see that it reduces query complexity in our two metrics, generating a high percentage of queries that can be resolved by BINSEC/ASE without a solver call, and the remainder is still less complex to solve.

B.5 Influence of Solver on Encoding Operators (RQ B4)

BINSEC/ASE inherits the native binding of BINSEC to the solver bitwuzla. Consequently, BINSEC/ASE results in a faster analysis with this native binding.

In this section, we consider the influence of the operator used in the forkless encoding and the solver used, considering z3, boolector and bitwuzla without the native binding.

RQ B4. What is the impact of the choice of the solver on the performance of the forkless encodings?

Goal. The goal of this experiment is to check whether the forkless encoding using the ite operator results in the fastest analysis independently of the solver used, or if different solvers favor different operators.

Protocol. We evaluate FASE without optimizations on the performance benchmark, with an attacker model able to perform 1 to 4 arbitrary data faults, with three different solvers (Bitwuzla without native binding, Boolector and z3), for four different encodings (Ite, Mul, And and Xor). We limit our experiment to 4 faults since some encodings start to experience timeouts after with the native binding that is faster than using external solvers.

Results. Analysis time results and average solving time per query are presented in Table B.13. None of these analyses experimented a timeout.

First, we can see that globally, Bitwuzla performs better than the other two solvers, even without the native binding, in analysis time and in average solving time per query. Bitwuzla is tuned for the specific SMT theory used by BINSEC (QF_ABV), while

Boolector and z3 are more generic in SMT theories, hence do not perform as well in this specific theory.

Bitwuzla still favors the Ite encoding, so does z3, while Boolector favors the Xor encoding. This shows the encoding used and the chosen solver have a big influence on the analysis time. For instance, with z3, the best-performing encoding yields an analysis x3.4 times faster than the worst for 1 fault. The best encoding of Bitwuzla results in an analysis x16.7 times faster than the slowest z3 encoding for 1 fault.

Table B.13: Analysis time and average solving time per query of arbitrary data faults with different solvers (RQ B4)

	1f	2f	3f	4f
Analysis time (s)				
Bitwuzla-Ite	10.4	38.7	130	358
Bitwuzla-Mul	13.8	56.7	252	979
Bitwuzla-And	14.1	59.6	311	1.26k
Bitwuzla-Xor	11.0	40.2	136	371
Boolector-Ite	25.0	125	423	1.22k
Boolector-Mul	38.8	198	717	2.21k
Boolector-And	40.4	219	836	2.4k
Boolector-Xor	23.5	113	377	1.05k
Z3-Ite	51.3	279	872	2.12k
Z3-Mul	147	761	1.97k	4.3k
Z3-And	154	790	2.03k	4.49k
Z3-Xor	174	877	2.12k	4.38k
Average solving time per query (s)				
Bitwuzla-Ite	0.065	0.052	0.055	0.054
Bitwuzla-Mul	0.089	0.074	0.088	0.087
Bitwuzla-And	0.09	0.087	0.087	0.107
Bitwuzla-Xor	0.071	0.056	0.066	0.06
Boolector-Ite	0.198	0.224	0.231	0.241
Boolector-Mul	0.28	0.294	0.301	0.32
Boolector-And	0.278	0.299	0.31	0.34
Boolector-Xor	0.183	0.202	0.207	0.214
Z3-Ite	0.273	0.305	0.324	0.341
Z3-Mul	0.663	0.791	0.751	0.74
Z3-And	0.688	0.801	0.773	0.75
Z3-Xor	0.707	0.834	0.772	0.747

Conclusion RQ B4. As SMT queries are a bottleneck for symbolic execution, it is important to select a solver tuned for the SMT theory used and to select operators to yield good analysis performance.

Appendix C

Instrumentation Details

This appendix chapter aims to provide more information about the way we instrumented programs of our benchmark for evaluation (Section 6.5).

Attacker Model. We preserve the attack goal written with an *assert* in each benchmark program and do not fault its computation. We consider here a user-defined number of arbitrary data faults.

General C Instrumentation. Programs are written in C. In Table C.1, we illustrate how each type of C statement is instrumented.

- The right-hand side of assignments are wrapped into a utility function *inject_AD*, injecting the fault;
- Conditional statements are expended. First, a new variable is created and holds the faulted conditional expression. It is this new variable that is inserted into the conditional statement;
- Return statements are instrumented similarly to assignments.

Table C.1: Instrumentation examples

C code	Instrumented C code
`int x = expr;`	`int x = inject_AD(expr);`
`if (cdt_expr) { \\... }`	`int cdt_1 = inject_AD(cdt_expr); if (cdt_1) { \\... }`
`return expr;`	`return inject_AD(expr);`

Instrumentation Settings. We add instrumentation settings as calls to instrumentation library functions from the main of the instrumented program as illustrated in Figure C.1 for a VerifyPIN program.

- We initialize instrumentation with the fault budget and whether we consider forkless or forking faults, by calling the *init_injection* function;
- The body of the $verifyPIN$ function if instrumented as presented above;
- A check constraining the total number of faults is inserted before the *assert* for forkless faults.

Instrumentation Library. The functions described in Figure C.2 are written in a separate file, serving as a library.

```
1 #define MAX_FAULTS 0
2 #define IS_FORKLESS false
3
4 int main(){
5     init_injection(MAX_FAULTS, IS_FORKLESS);
6     initialize();
7     verifyPIN();
8     int o = oracle();
9     fault_condition();
10    assert(o);
11    return 0;
12 }
```

Figure C.1: Instrumented VerifyPIN main function

- The function *init_injection* sets global variables holding the analysis injection parameters;
- The function *fault_condition* adds the fault budget constraint for forkless faults;
- The *inject_AD* function first creates a new symbolic boolean variable for the fault activation and adds it to the *sum* variable. If we consider forkless faults, a new symbolic variable is created for the effect of the arbitrary data fault, and a forkless fault encoding is returned, taking the place of the original expression. The forkless encoding chosen uses a And operator. In the case of forking faults, the fault budget is checked first, then the control flow is split depending on the symbolic value of *b*. If the fault happens, a new symbolic value is returned, otherwise, the original expression is returned.

Instrumentation Extension. This instrumentation framework has been written for the evaluation of arbitrary data faults. However, it can be extended to support more:

- It is possible to extend this instrumentation framework to other data fault models or other forkless encodings simply by modifying the fault encoding in the *inject_AD* function;
- Fault models other than data faults can also be implemented based on this template, they may require slightly different C instrumentation than what is presented in Table C.1 though;
- We believe EDS optimization can be implemented as well in this framework. Some work or rethinking of optimizations might be needed before implementing other optimizations.

```
1 int sum, max_faults;
2 bool is_forkless;
3
4 void init_injection(int max, bool ib) {
5     sum = 0;
6     max_faults = max;
7     is_forkless = ib;
8 }
9
10 bool fault_condition() {
11     if (is_forkless)    {
12         klee_assume(sum <= max_faults);
13     }
14     return true;
15 }
16
17 int inject_AD(int x) {
18     int b;
19     klee_make_symbolic(&b, sizeof(int), "b");
20     klee_assume(b <= 1);
21     klee_assume(b >= 0);
22     sum += b;
23
24     if (is_forkless)    {
25         int non_det;
26         klee_make_symbolic(&non_det, sizeof(int),
27                                          "non_det");
28         klee_assume(non_det != 0);
29         return x + ((-b) & non_det);
30     }
31     else {
32         klee_assume(sum <= max_faults);
33         if (b){
34             int non_det;
35             klee_make_symbolic(&non_det, sizeof(int),
36                                              "non_det");
37             klee_assume(x != non_det);
38             return non_det;
39         }
40         return x;
41     }
42 }
```

Figure C.2: Instrumentation utilitary functions

Bibliography

[AAE+20] ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales, and Trusted Labs. Inter-cesti: Methodological and technical feedbacks on hardware devices evaluations. In *SSTIC 2020, Symposium sur la sécurité des technologies de l'information et des communications*, 2020. 133

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The em side—channel (s). In *Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4*, pages 29–45. Springer, 2003. 17

[ABEL09] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009. 19

[AC04] Martín Abadi and Véronique Cortier. Deciding knowledge in security protocols under equational theories. In *International Colloquium on Automata, Languages, and Programming*, pages 46–58. Springer, 2004. 64

[ACD+22] Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle, and Paolo Maistri. Variable-length instruction set: Feature or bug? In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 464–471. IEEE, 2022. 60

[AKS06] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*, pages 225–242. Springer, 2006. 17

[ASA+15] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48:44–57, 2015. 23, 64

[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 2007)*, pages 92–102. IEEE, 2007. 20, 46

[BB09] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*, pages 174–177. Springer, 2009. 71

[BBB+10] Alessandro Barenghi, Guido M Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. Low voltage fault attacks to aes. In *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 7–12. IEEE, 2010. 20

[BBC+14] Maël Berthier, Julien Bringer, Hervé Chabanne, Thanh-Ha Le, Lionel Rivière, and Victor Servant. Idea: embedded fault injection simulator on smartcard. In *International Symposium on Engineering Secure Software and Systems*, pages 222–229. Springer, 2014. 3, 35, 62, 63, II

[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(11):3056–3076, 2012. 19, 20

[BBY17] Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring abstract interpreters through state and value abstractions. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 112–130. Springer, 2017. 61, 62

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018. 30

[BCDK14] Sébastien Bardin, Omar Chebaro, Mickaël Delahaye, and Nikolai Kosmatov. An all-in-one toolkit for automated white-box testing. In *International Conference on Tests and Proofs*, pages 53–60. Springer, 2014. 36, 61

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 351–366. Springer, 2008. 33

[BCL14] Gergei Bana and Hubert Comon-Lundh. A computationally complete symbolic attacker for equivalence properties. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 609–620, 2014. 3, 64

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer, 2004. 2, I

[BCN+17] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017. 19, 64

[BCR+19] Cataldo Basile, Daniele Canavese, Leonardo Regano, Paolo Falcarin, and Bjorn De Sutter. A meta-model for software protections and reverse engineering attacks. *Journal of Systems and Software*, 150:3–21, 2019. 64

[BD12] Leyla Bilge and Tudor Dumitraş. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 833–844, 2012. 13

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997. 128

[BDL01] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of cryptology*, 14:101–119, 2001. 20

[BDM17] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded dse: targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 633–651. IEEE, 2017. 70

[BECN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006. 19

[BEMP20] Etienne Boespflug, Cristian Ene, Laurent Mounier, and Marie-Laure Potet. Countermeasures optimization in multiple fault-injection context. In *2020 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 26–34. IEEE, 2020. 130

[Ber05] Daniel J Bernstein. Cache-timing attacks on aes. 2005. 17

[BFP19] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the glitch: optimizing voltage fault injection attacks. *IACR transactions on cryptographic hardware and embedded systems*, pages 199–224, 2019. 20

[BG22] Sébastien Bardin and Guillaume Girol. A quantitative flavour of robust reachability. *arXiv preprint arXiv:2212.05244*, 2022. 61

[BGM13] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 122–131. IEEE, 2013. 2, I

[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 *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 105–114. IEEE, 2011. 20

[BHE+19] Jean-Baptiste Bréjon, Karine Heydemann, Emmanuelle Encrenaz, Quentin Meunier, and Son-Tuan Vu. Fault attack vulnerability assessment of binary code. In *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*, pages 13–18, 2019. 3, 35, 62, 63, II

[BHL+11] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The bincoa framework for binary code analysis. In *International Conference on Computer Aided Verification*, pages 165–170. Springer, 2011. 70

[BKC14] Sébastien Bardin, Nikolai Kosmatov, and François Cheynier. Efficient leveraging of symbolic execution to advanced coverage criteria. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 173–182. IEEE, 2014. 36

[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 *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–6, 2018. 20

[BMPM13] Stephen P Buchner, Florent Miller, Vincent Pouget, and Dale P McMorrow. Pulsed-laser testing for single-event effects investigations. *IEEE Transactions on Nuclear Science*, 60(3):1852–1875, 2013. 19

[BR10] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):1–84, 2010. 34

[BRT+18] Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnauld Michelizza, and Jérémy Lefaure. Wookey: Usb devices strike back. *Proceedings of SSTIC*, 2018. 132

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology—CRYPTO'97: 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings 17*, pages 513–525. Springer, 1997. 20

[BSGD09] Shivam Bhasin, Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. Security evaluation of different aes implementations against practical setup time violation attacks in fpgas. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 15–21. IEEE, 2009. 20

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018. 30

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19:7–34, 2001. 2, 29, I

[CC2] Common criteria for information technology security evaluation. part 3: Security assurance components, cc:2022, revision 1. https://www.commoncriteriaportal.org/files/ccfiles/CC2022PART5R1.pdf. 12, 15

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 14*, pages 21–30. Springer, 2005. 2, I

[CCG13] Maria Christofi, Boutheina Chetali, and Louis Goubin. Formal verification of an implementation of crt-rsa vigilant's algorithm. In *PROOFS workshop: pre-proceedings*, volume 28, 2013. 3, 35, 62, 63, II

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008. 62

[CdFB21] Hervé Chauvet, François de Ferrière, and Thomas Bizet. Software fault injection for secswift qualification, 2021. 130

[CDFG18] Sébastien Carré, Matthieu Desjardins, Adrien Facon, and Sylvain Guilley. Openssl bellcore's protection helps fault attack. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 500–507. IEEE, 2018. 3, 35, 62, 63, II

[CDSLN20] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. Profipy: Programmable software fault injection as-a-service. In *2020 50th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 364–372. IEEE, 2020. 36

[Cer01] Iliano Cervesato. The dolev-yao intruder is the most powerful attacker. In *16th Annual Symposium on Logic in Computer Science—LICS*, volume 1, pages 1–2. Citeseer, 2001. 3

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003. 2, I

[CGMN12] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143, 2012. 18

[CGP+08] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008. 30

[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: 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11–12, 2021, Revised Selected Papers*, pages 151–166. Springer, 2022. 19

[Cla97] Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer, 1997. 29

[CMD+19] Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–10. IEEE, 2019. 19

[CN13] Domenico Cotroneo and Roberto Natella. Fault injection for software certification. *IEEE Security & Privacy*, 11(4):38–45, 2013. 36

[Cou21] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021. 28

[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, 2021. 20

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013. 2, 29, 30, I

[CTB+17] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Bart Coppens, Bjorn De Sutter, Paolo Falcarin, and Marco Torchiano. How professional hackers understand protected code while performing attack tasks. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 154–164. IEEE, 2017. 64

[CTB+19] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empirical Software Engineering*, 24(1):240–286, 2019. 64

[CW96] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996. 27

[CWP+00] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE, 2000. 18

[Dan21] Lesly-Ann Daniel. *Symbolic binary-level code analysis for security. Application to the detection of microarchitectural timing attacks in cryptographic code*. PhD thesis, Université Côte d'Azur, 2021. ix, 27, 28

[DB15] Adel Djoudi and Sébastien Bardin. Binsec: Binary code analysis with low-level regions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 212–217. Springer, 2015. 70

[DBP23] Soline Ducousso, Sébastien Bardin, and Marie-Laure Potet. Adversarial reachability for program-level security analysis. *Programming Languages and Systems LNCS 13990*, page 59, 2023. 4, 38, 65, III

[DBR20] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038. IEEE, 2020. 31, 61, 70

[DBR21] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse. In *NDSS*, 2021. 70

[DBT+16] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *SANER*, 2016. 4, 70

[DDCS+14] Jean-Max Dutertre, Stephan De Castro, Alexandre Sarafianos, Noémie 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 *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2014. 19

[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 *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15. IEEE, 2012. 20

[dF21] François de Ferrière. Software countermeausres in the llvm risc-v compiler, 2021. 130

[DHM+23] Mathieu Dumont, Kevin Hector, Pierre-Alain Moellic, Jean-Max Dutertre, and Simon Pontié. Evaluation of parameter-based attacks against embedded neural networks with laser injection. *arXiv preprint arXiv:2304.12876*, 2023. 131

[DLG21] Zhen Yu Ding and Claire Le Goues. An empirical study of oss-fuzz bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142. IEEE, 2021. 26

[DLRA15] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–29, 2015. 18

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008. 71

[DMM+13] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. Electromagnetic glitch on the aes round counter. In *Constructive Side-Channel Analysis and Secure Design: 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers 4*, pages 17–31. Springer, 2013. 20

[DOL+10] Amine Dehbaoui, Thomas Ordas, Victor Lomné, Philippe Maurine, Lionel Torres, and Michel Robert. Incoherence analysis and its application to time domain em analysis of secure circuits. In *2010 Asia-Pacific International Symposium on Electromagnetic Compatibility*, pages 1039–1042. IEEE, 2010. 17

[DPdC+15] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas, and Jessy Clédière. From code review to fault injection attacks: Filling the gap using fault model inference. In *International conference on smart card research and advanced applications*, pages 107–124. Springer, 2015. 34, 60, 64

[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 *International Conference on Computer Safety, Reliability, and Security*, pages 3–11. Springer, 2016. ix, 15, 16, 20, 86, 87, 105

[DRPR19] Jean-Max Dutertre, Timothé Riom, Olivier Potin, and Jean-Baptiste Rigaud. Experimental analysis of the laser-induced instruction skip fault model. In *Secure IT Systems: 24th Nordic Conference, NordSec 2019, Aalborg, Denmark, November 18–20, 2019, Proceedings 24*, pages 221–237. Springer, 2019. 19, 20

[Dul17] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2017. 129, 130

[DY83] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983. 64

[Fac] Facebook. Infer static analyzer. https://fbinfer.com/. 2, I

[FDBL18] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing. In *LPAR*, pages 363–380, 2018. 70

[FMB+16] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, pages 1–12, 2016. 31

[FR08] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 2008. 65

[GDTM21a] Joseph Gravellier, Jean-Max Dutertre, Yannick Teglia, and Philippe Loubet Moundi. Faultline: Software-based fault injection on memory transfers. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 46–55. IEEE, 2021. 21

[GDTM21b] Joseph Gravellier, Jean-Max Dutertre, Yannick Teglia, and Philippe Loubet Moundi. Sideline: How delay-lines (may) leak secrets from your soc. In *Constructive Side-Channel Analysis and Secure Design: 12th International Workshop, COSADE 2021, Lugano, Switzerland, October 25–27, 2021, Proceedings 12*, pages 3–30. Springer, 2021. 21

[GFB21] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not all bugs are created equal, but robust reachability can tell the difference. In *International Conference on Computer Aided Verification*, pages 669–693. Springer, 2021. 31, 61

[GHHR23] Antoine Gicquel, Damien Hardy, Karine Heydemann, and Erven Rohou. Samva: Static analysis for multi-fault attack paths determination. In *Constructive Side-Channel Analysis and Secure Design: 14th International Workshop, COSADE 2023, Munich, Germany, April 3–4, 2023, Proceedings*, pages 3–22. Springer, 2023. 62, 63

[Gir05] Christophe Giraud. Dfa on aes. In *Advanced Encryption Standard–AES: 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers 4*, pages 27–41. Springer, 2005. 20

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005. 30

[GLM12] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012. 2, I

[GMA22] Aakash Gangolli, Qusay H Mahmoud, and Akramul Azim. A systematic review of fault injection attacks on iot systems. *Electronics*, 11(13):2023, 2022. 64

[God11] Patrice Godefroid. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 258–269, 2011. 31

[God20] Patrice Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 63(2):70–76, 2020. 28

[GST17] Daniel Genkin, Adi Shamir, and Eran Tromer. Acoustic cryptanalysis. *Journal of Cryptology*, 30:392–443, 2017. 17

[GSV03] Bharat Goyal, Sriranjani Sitaraman, and S Venkatesan. A unified approach to detect binding based race condition attacks. In *Int'l Workshop on Cryptology & Network Security (CANS)*, page 16, 2003. 23

[GWJL20] Thomas Given-Wilson, Nisrine Jafri, and Axel Legay. Combined software and hardware fault injection vulnerability detection. *Innovations in Systems and Software Engineering*, 16(2):101–120, 2020. 3, 35, 62, 63, II

[GWJLL17] 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 *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 293–300. IEEE, 2017. 3, 35, 62, 63, II

[GWL20] Thomas Given-Wilson and Axel Legay. Formalising fault injection and countermeasures. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–11, 2020. 65

[HH19] Reiner Hähnle and Marieke Huisman. Deductive software verification: from pen-and-paper proofs to industrial tools. *Computing and Software Science: State of the Art and Perspectives*, pages 345–373, 2019. 27

[HS14] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *Smart Card Research and Advanced Applications: 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers 12*, pages 219–235. Springer, 2014. 20

[HSP20] 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, 2020. 34

[HTS+17] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 249–258. IEEE, 2017. 36

[JGH+22] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. Evocatio: Conjuring bug capabilities from a single poc. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1599–1613, 2022. 64

[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(3):361–372, 2014. 21

[KHF+20] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020. 22

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. 30

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, pages 388–397. Springer, 1999. 17

[KKBC12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012. 33

[KKJ+21] Kyounggon Kim, Jun Seok Kim, Seonghoon Jeong, Jo-Hee Park, and Huy Kang Kim. Cybersecurity for autonomous vehicles: Review of attacks and defense. *Computers & Security*, 103:102150, 2021. 11

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal aspects of computing*, 27(3):573–609, 2015. 2, 61, 62, I

[Koc96] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996. 17

[KP05] Alfred Koelbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33:645–666, 2005. 33

[KSV13] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, 2013. 19

[Lan22] Julien Lancia. Detecting fault injection vulnerabilities in binaries with symbolic execution. In *2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–8. IEEE, 2022. 3, 35, 62, 63, II

[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. 19

[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 *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 561–564. IEEE, 2018. 60, 64

[LBDPP19] 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 *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 252–255. IEEE, 2019. 60

[LFBP21] Guilhem Lacombe, David Feliot, Etienne Boespflug, and Marie-Laure Potet. Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. In *PROOFS WORKSHOP (SECURITY PROOFS FOR EMBEDDED SYSTEMS)*, 2021. ix, 3, 35, 61, 62, 63, 133, 135, 136, 137, II

[LH07] Daniel Larsson and Reiner Hähnle. Symbolic fault injection. In *International Verification Workshop (VERIFY)*, volume 259, pages 85–103. Citeseer, 2007. 36

[LHGD18] Hoang M Le, Vladimir Herdt, Daniel Große, and Rolf Drechsler. Resilience evaluation via symbolic fault injection on intermediate code. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 845–850. IEEE, 2018. 36, 62, 105

[LM18] Yangdi Lyu and Prabhat Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2:33–50, 2018. 17

[LRV+22] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022. 2, I

[LWH+20] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020. 29

[MBK+18] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loïc Correnson. Time to clean your test objectives. In *Proceedings of the 40th International Conference on Software Engineering*, pages 456–467, 2018. 36

[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 *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88. Ieee, 2013. 20

[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 *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–7. IEEE, 2020. 20

[MK19] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019. 21

[MKP22] Thibault Martin, Nikolai Kosmatov, and Virgile Prevosto. Verifying redundant-check based countermeasures: a case study. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1849–1852, 2022. 61, 62, 63, 137

[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. 22

[NHH+17] Shoei Nashimoto, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki. Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of Cryptographic Engineering*, 7:35–46, 2017. 20

[NP20] Aina Niemetz and Mathias Preiner. Bitwuzla at the SMT-COMP 2020. *CoRR*, abs/2006.01621, 2020. 71

[OGSM15] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. Em injection: Fault model and locality. In *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 3–13. IEEE, 2015. 20

[O'H19] Peter W O'Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019. 64

[PCHR20] Pierre-Yves Péneau, Ludovic Claudepierre, Damien Hardy, and Erven Rohou. Nop-oriented programming: Should we care? In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 694–703. IEEE, 2020. 19

[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, 2019. 20

[PIK+18] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges,

and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 47–53. IEEE, 2018. 36

[PLFP19] Lucas Palazzi, Guanpeng Li, Bo Fang, and Karthik Pattabiraman. A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 151–162. IEEE, 2019. 36

[PM10] Mike Papadakis and Nicos Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 121–130. IEEE, 2010. 36

[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 *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 213–222. IEEE, 2014. 3, 35, 62, 63, 124, II

[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. IEEE, 2008. 36

[PRBL14] Maxime Puys, Lionel Riviere, Julien Bringer, and Thanh-ha Le. High-level simulation for multiple fault injection evaluation. In *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance*, pages 293–308. Springer, 2014. 128

[QS02] Jean-Jacques Quisquater and David Samyde. Eddy current for magnetic analysis with active sensor. In *Proceedings of eSMART*, volume 2002, 2002. 20

[RBB+19] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. Get rid of inline assembly through verification-oriented lifting. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 577–589. IEEE, 2019. 70

[RBB+21] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Matthieu Lemerre, Laurent Mounier, and Marie-Laure Potet. Interface compliance of inline assembly: Automatically check, patch and refine. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1236–1247. IEEE, 2021. 70

[RBSG22] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Guneysu. Revisiting fault adversary models–hardware faults in theory and practice. *IEEE Transactions on Computers*, 2022. 64

[RG14] Pablo Rauzy and Sylvain Guilley. A formal proof of countermeasures against fault injection attacks on crt-rsa. *Journal of Cryptographic Engineering*, 4(3):173–185, 2014. 3, 35, 62, 63, II

[RGB+16] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security symposium*, volume 25, pages 1–18, 2016. 21

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953. 27

[RNR+15] Lionel Riviere, 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 *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2015. 19, 20

[RY20] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020. 28

[SD15] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15:71, 2015. 21

[Sen07] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007. 29

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005. 30

[SNK+12] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. Simple photonic emission analysis of aes: photonic side channel analysis for the rest of us. In *Cryptographic Hardware and Embedded Systems–CHES 2012: 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings 14*, pages 41–57. Springer, 2012. 17

[TAC+22] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. Exploration of fault effects on formal risc-v microarchitecture models. In *2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 73–83. IEEE, 2022. 60, 61, 62, 63, 98

[TM17] Niek Timmers and Cristofaro Mune. Escalating privileges in linux using voltage fault injection. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8. IEEE, 2017. 20

[TS16] Niek Timmers and Albert Spruyt. Bypassing secure boot using fault injection. *Black Hat Europe*, 2016, 2016. 20

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Clkscrew: Exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, volume 2, pages 1057–1074, 2017. 22

[Van22] Julien Vanegue. Adversarial logic. In *Static Analysis: 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5–7, 2022, Proceedings*, pages 422–448. Springer, 2022. 64

[VBMS+20] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72. IEEE, 2020. 23

[VKS11] Ingrid Verbauwhede, Dusko Karaklajic, and Jorn-Marc Schmidt. The fault attack jungle-a classification model to guide you. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 3–8. IEEE, 2011. 20

[VTM+17] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adele Morisset, and Sébastien Ermeneux. Laser-induced fault injection on smartphone bypassing the secure boot. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 41–48. IEEE, 2017. 19, 20

[Woo] https://github.com/wookey-project. accessed july 2021. 132

[WP05] Jinpeng Wei and Calton Pu. Tocttou vulnerabilities in unix-style file systems: An anatomical study. In *FAST*, volume 5, pages 12–12, 2005. 23

[WTSS13] Stefan Winter, Michael Tretter, Benjamin Sattler, and Neeraj Suri. simfi: From single to simultaneous software fault injections. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013. 36

[WZ18] Maximilian Wohrer and Uwe Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8. IEEE, 2018. 10

[XTDHS09] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. IEEE, 2009. 33

[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 *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58. IEEE, 2016. 20

[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 *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014. 20

[ZGWL+21] Igor Zavalyshyn, Thomas Given-Wilson, Axel Legay, Ramin Sadre, and Etienne Riviere. Chaos duck: A tool for automatic iot software fault-tolerance analysis. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 46–55. IEEE, 2021. 36