Program-Structure–Guided Reduction of the Execution Time of Fault-Injection Campaigns on the ISA Layer

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades

DOKTOR-INGENIEUR
(abgekürzt: Dr.-Ing.)
genehmigte Dissertation

von Herrn
Oskar Pusz, M.Sc.
geboren am 21. November 1990
in Hannover, Deutschland

2024

1. Referent	**Prof. Dr.-Ing. habil. Daniel Lohmann**
2. Referent	**Prof. Dr.-Ing. Horst Schirmeier**
Tag der Promotion	**19.08.2024**

Abstract

Due to shrinking transistor structure sizes and operating voltages, hardware becomes more susceptible to transient hardware faults. In the domain of safety-critical systems, fault injection campaigns on the instruction-set–architecture layer have become a widespread approach to assess the resilience of a system concerning this kind of fault. Full fault-injection campaigns are an approach to systematically assess the reliability of a system and the effectiveness of implemented software-based hardening techniques on fixed hardware.

A straightforward fault-injection campaign may result in practically unrealizable runtimes, especially when aiming for a comprehensive and complete reliability analysis of the system under test. Established acceleration methods are common to either reduce the number of necessary fault injections or speed up individual injections, ultimately decreasing the overall runtime of the whole campaign.

However, despite the effectiveness of these established methods, the runtimes may still be infeasible, the campaign results lack precision, or the focus might be limited to specific aspects of the system under test only.

This dissertation introduces three new approaches to handling these challenges. The approaches use extracted program structures of the executed software, tailored to the running program independent from system behaviors under evaluation.

The first approach extracts the data flow and instruction semantics to utilize instructions' propagation and masking effects through a data flow graph. Compared to the ground truth method, my data-flow-sensitive acceleration method significantly reduces the number of necessary injections for a comprehensive reliability analysis by up to 18.4 percent precisely.

The second approach utilizes extracted dynamic jump addresses to represent the control flow, partitioning the program's execution into temporal segments. These fault-space regions operate as distinct entities, each with its data flow potentially flowing from one to the next. Injecting the traversing data flows and approximating their results to the other non-traversing data flows leads to an injection reduction of up to 77.5 percent system-wide, accompanied by an approximation error of only 2 percent and a strong locality of the results.

The last contribution focuses on accelerating individual injections that do not lead to the termination of systems, thus, reaching a fixed timeout threshold. This work presents an analysis of timeouts in this context and initial approaches to predict such timeouts during runtime. The final part of this contribution is the timeout detector, ACTOR. This detector uses autocorrelation to detect whether patterns exist in the program's taken jumps, thereby approximating whether the program is in a loop. ACTOR can achieve end-to-end campaign accelerations of up to 27.6 percent through timeout predictions in individual injections. Thereby, the absolute prediction error is always less than 0.5 percent.

The methods developed in this work expand the overall portfolio of potential acceleration methods in the fault-injection community. These generically designed methods, implemented and evaluated in the instruction-set–architecture layer, can also be conceptually applied to other system layers. They offer versatility and are seamlessly combinable with each other and established acceleration methods.

Kurzfassung

Aufgrund immer kleinerer Transistorgrößen und Betriebsspannungen wird Hardware immer anfälliger für transiente Hardwarefehler. Im Bereich sicherheitskritischer Systeme haben sich Fehlerinjektionskampagnen auf der Ebene der Befehlssatzarchitektur zu einem weit verbreiteten Ansatz entwickelt, um die Funktionssicherheit eines Systems im Hinblick auf diese Art von Fehlern zu bewerten. Vollständige Fehlerinjektionskampagnen sind ein Ansatz zur systematischen Bewertung der Zuverlässigkeit eines Systems einerseits und der Effektivität von implementierten softwarebasierten Härtungstechniken auf im Vorhinein spezifizierter Hardware andererseits.

Eine naiv ausgeführte Fehlerinjektionskampagne führt durch die schiere Menge an Injektionen allerdings schnell zu praktisch nicht realisierbaren Laufzeiten, insbesondere wenn eine umfassende und vollständige Funktionssicherheitsanalyse des zu testenden Systems angestrebt wird. Um dem entgegenzuwirken, sind etablierte Beschleunigungsmethoden üblich, die entweder die Anzahl notwendiger Fehlerinjektionen reduzieren oder individuelle Injektionen beschleunigen, was letztlich die Gesamtlaufzeit einer gesamten Kampagne reduziert.

Es kann jedoch vorkommen – trotz der Effektivität von etablierten Methoden –, dass die Kampagnen immer noch zu lange Laufzeiten besitzen, dass die Ergebnisse nicht präzise genug sind oder dass der Fokus nur auf bestimmte Aspekte des zu testenden Systems beschränkt ist.

Diese Dissertation stellt drei neue Ansätze vor, die darauf abzielen, diese Herausforderungen zu bewältigen. Dazu verwenden sie Programmstrukturen, die aus gegebener Software extrahiert werden. Diese sind auf das laufende Programm zugeschnitten und unabhängig vom zu evaluierenden Systemverhalten.

Der erste Beitrag extrahiert Datenflüsse und Instruktionssemantik, um Propagations- und Maskierungseffekte von Instruktionen über einen Datenflussgrafen zu nutzen. Im Vergleich zur Referenzmethode reduziert meine datenflusssensitive Beschleunigungsmethode die Anzahl der notwendigen Injektionen für eine vollständige Funktionssicherheitsanalyse präzise um bis zu 18,4 Prozent.

Der zweite Beitrag nutzt extrahierte dynamische Sprungadressen als Repräsentanten des Kontrollflusses und partitioniert die Programmausführung in zeitliche Segmente. Diese Segmente heißen Fehlerraum-Regionen und agieren als eigenständige Entitäten, von denen jede ihren eigenen Datenfluss hat, der potenziell von einer zur nächsten fließt. Die Injektion der austretenden Datenflüsse und die Approximation ihrer Ergebnisse auf die anderen Datenflüsse, die eine Region nicht verlassen, führt zu einer systemweiten Reduktion der Injektionen von bis zu 77,5 Prozent mit einem Approximationsfehler von nur 2 Prozent und einer starken Lokalität der Ergebnisse.

Der letzte Beitrag konzentriert sich auf die Beschleunigung individueller Injektionen, die nicht zur Terminierung eines Systems führen, und daher nach einer festgelegten Frist (Timeout) beendet werden. Dazu präsentiere ich eine Analyse von Timeouts in diesem Zusammenhang und erste Ansätze zur Vorhersage solcher während der Laufzeit. Der finale Teil dieses Beitrags ist der Timeout-Detektor ACTOR. Dieser Detektor nutzt Autokorrelation, um Muster in den genommenen Sprüngen des Programms zu erkennen, um damit approximativ zu bestimmen, ob sich das Programm in einer Schleife befindet. ACTOR ist in der Lage, durch Timeout-Vorhersagen für individuelle Injektionen Ende-zu-Ende Kampagnenlaufzeitbeschleunigungen von bis zu 27,6 Prozent zu erreichen. Dabei ist der absolute Fehler in den Vorhersagen durchgehend bei unter 0,5 Prozent.

Die in dieser Arbeit entwickelten Methoden erweitern das Gesamtportfolio potenzieller Beschleunigungsmethoden in der Forschungsgemeinschaft der Fehlerinjektion. Diese generisch konzipierten Methoden, die in der Ebene der Befehlssatzarchitektur implementiert und evaluiert wurden, können konzeptionell auch auf andere Systemebenen angewendet werden. Sie sind vielseitig einsetzbar

und lassen sich sowohl untereinander als auch mit etablierten Beschleunigungsmethoden nahtlos kombinieren.

Danksagungen

Mein Weg zu dieser Dissertation ist mit vielen tollen Menschen gepflastert, denen ich an dieser Stelle danken möchte. Jeder für sich hat innerhalb meiner Zeit als HiWi und wissenschaftlicher Mitarbeiter einen Teil davon auf unterschiedlichstem Wege beigetragen.

Ioannis akquirierte mich beim Fachgebiet System- und Rechnerarchitektur (SRA) der Leibniz Universität Hannover (LUH) als studentische Hilfskraft und startete damit die Kette an Ereignissen, die mich durch die Zeit am SRA begleitet haben. Danke Dir für Dein Vertrauen, das Du damals in mich hattest, Dich in Deinen Forschungsfragen zu unterstützen.

Eine lange Zeit als Tutor und HiWi war ich bei Sebastian, der auch meine beiden Abschlussarbeiten beide betreut hat. Danke, Sebastian, für Deine aufopfernde Betreuung meiner Bachelor- und Masterarbeit, wie auch für die tolle Erfahrung an “Mantella” und “Hugo” zu arbeiten, für die inspirierenden Diskussionen und meine ersten Erfahrungen in der Lehre. Auch Dir, unserem “Fachgebietsinventar” Kiechle, danke ich für die Unterstützung über die Jahre, sei es in der Lehre und Forschung wie auch mit Sebastian zusammen für unser gemeinsames wildes Abenteuer als Aussteller auf der Hannover Messe.

Während meines Praktikums und kurz vor meiner Masterarbeit lernte ich meinen dortigen Betreuer für Studierende Dennis kennen. Er als Promovierter der LUH hat mir während diverser Mittagspausen das wissenschaftliche Arbeiten erläutert und schmackhaft gemacht. Auch wenn es Dir nicht bewusst war, Dennis, aber Du hast den ursprünglichen Samen gesät, der mein Interesse an der wissenschaftlichen Arbeit geweckt hat und dafür danke ich Dir.

Jürgen, Du hast mich während der Anfertigung meiner Masterarbeit damals auf dem Fahrrad auf dem Weg nach Hause noch gefragt, ob ich denn nicht Lust hätte zu promovieren. Damit hast Du zeitlich passend die Weichen gestellt, die mich kurz vor Fertigstellung der Masterarbeit zum wissenschaftlichen Mitarbeiter gemacht haben. Danke Dir dafür, für die Zusammenarbeit in Forschung und Lehre, wie auch für alle nicht-universitären Annehmlichkeiten Deinerseits, sei es für alle Leckereien aus dem Frankenland oder Deinem “Zaubertrank”.

Ich danke ebenfalls vielmals Herrn Christian Müller-Schloer für die Zeit, die ich unter Ihnen als Fachgebietsleiter erleben durfte. Danke für die inspirierenden Gespräche zu meiner Anfangszeit als wissenschaftlicher Mitarbeiter, die Geheimtipps für Wanderstrecken und dafür, dass Sie mich schlussendlich als Mitarbeiter eingestellt haben.

Ich danke auch den zu dieser Zeit am SRA anwesenden Doktoranden Yvonne, Henning, Jan und Lukas, die jeder für sich auf unterschiedliche Weise meine Zeit am SRA bereichert haben.

Mit dem Wechsel des Fachgebietsleiters kam mein Doktorvater Daniel ans SRA. Daniel, Du brachtest ein neues Forschungsgebiet mit und ich danke Dir für Dein Vertrauen in mich als Deinen “geerbten” Doktoranden wie auch für die stetigen Begeisterungsschübe, mich für mir unbekannte Thematiken zu begeistern. Ich danke Dir, dass Du mich auf dem Weg begleitet hast, für die lehrreiche Zusammenarbeit an Forschungsfragen, wie auch die aktive Mitarbeit an meinen Veröffentlichungen, die im Endeffekt zu dieser Dissertation geführt haben. Ich danke Dir auch für die Korrekturlesung und Deinen Dienst als Erstgutachter meiner Dissertation.

Ebenfalls danken möchte ich Horst, der sich dazu bereit erklärt hat, als Zweitgutachter meiner Dissertation zu fungieren. Horst, ich danke Dir für Dein Tool FAIL*, das meine Forschungsarbeit gestützt hat, wie auch für die fruchtbaren Diskussionen rund um das Thema Fehlerinjektionen, die meine Arbeit bereichert haben, wie auch die Vorträge für meine Studierenden im Rahmen von Seminaren.

Vielmals danken möchte ich an dieser Stelle auch meinem damaligen Kollegen Christian. Ich danke Dir, Christian, für all Deine konstruktive Zusammenarbeit und Deine passionserfüllten Beiträge

zu meinen Themen. Du hattest hervorragende Einwürfe zu meinen Themen und bist damit ebenfalls zu einem Teil am Erfolg meiner Arbeit beteiligt.

Vielen Dank auch an meine Abschlussarbeitenden Zena, Daniel K., Felix, Jannis und Tim, die jeweils mit ihrem Beitrag meine Veröffentlichungen positiv beeinflusst haben.

Rechenintensive Forschung benötigt starke Hardware, die ich lange Zeit dem fachgebietsinternen Cluster zu verdanken habe. Ich danke Dir, Lars M., für Deinen Dienst als unseren Techniker, die Betreuung der Infrastruktur und Server wie auch um das Kümmern aller technischen Belange rund um meine Arbeit.

Ein Dankeschön auch an Dich, Monika, unsere Sekretärin und der Engel der Verwaltung des SRA. Du hast Dich in meiner Zeit um so viel Formelles gekümmert, hast immer eine gute Idee gehabt Dinge zu organisieren und nimmst damit viel Arbeit ab.

Danke auch an all meine "Doktorgeschwister" oder sonstigen Leidensgenossen, die mich während meiner Zeit als Doktorand begleitet haben: Alexander, Andreas K., Andreas Z., Björn, Christian, Christoph, Dominik, Florian K., Florian R., Gerion, Lars W., Ralf, Romeo, Stefan, Tim, Tobias und Valentin. Danke für die enge und familiäre Zusammenarbeit in Lehre, Forschung wie auch für die tollen Momente nach Feierabend oder auf Ausflügen, die die Zeit am SRA neben den Unmengen Kilos an Leibniz-Keks–Bruchware weiter versüßt hat.

Ebenfalls bedanke ich mich bei diversen Gastforschenden und Studierenden in unserem Fachgebiet für die interessanten und inspirierenden Gespräche zu allerlei spannenden Themen der Informatik.

Ich bedanke mich an dieser Stelle auch bei allen beteiligten Korrekturlesenden, die mit ihrem Beitrag diese Dissertation nochmals zu einer besseren gemacht haben. Ich danke Euch, Daniel, Daniela, Maximilian, Tim und Gerion.

Vielen Dank an die tollen Leute in meinem Leben außerhalb des SRAs, die mich stets motiviert und unterstützt haben. Ich danke ebenfalls für diverse Aufmerksamkeiten, wie die "Physiker-Tüte einer noblen, intellektuellen Sorte" und das Verständnis aller, dass ich mich von allem im Alltag abkapseln musste.

Besonderen Dank auch an meine Partnerin Daniela, die mich während der Zeit der Anfertigung der Dissertation tatkräftig unterstützt hat. Danke Dir, Daniela, für jeden Gedanken und für jede einzelne Stunde Deiner Zeit, die Du für mich investiert hast und für meine "Dissertationstasse".

Zuallerletzt möchte ich mich hier bei meiner Mutter Marta bedanken. Mama, letztendlich hast Du mit einfach allem in unserem Leben dafür gesorgt, dass ich genau diesen Text hier verfassen darf. Dies ist auch Dein Verdienst, dass ich das erreichen durfte. Danke, Mama!

Hannover, August 2024

Table of Contents

1

Introduction

A problem is a chance for you to do your best.

EDWARD KENNEDY "DUKE" ELLINGTON (1899–1974)

1.1 Motivation

Semiconductor-based integrated circuits are ubiquitous today, influencing various aspects of daily life through devices such as cell phones and household appliances. Embedded systems, such as those found in automobiles, execute essential functions like assistance or multimedia systems, forming complex computer networks with multiple control units. The continuous miniaturization of hardware structures [Com22] and the declining unit costs [Sch97] have also impacted industries in domains like aviation and space flight. In *safety-critical* domains like those mentioned above, ensuring *reliability* is paramount. Reliability entails a system consistently fulfilling its *externally observable* purpose within specified time frames, free from failures, regardless of prevailing circumstances [Avi+04]. If a system cannot maintain reliability and eventually will fail, it is *safe* if it can prevent catastrophic consequences [Avi+04].

This prevalence stems from the ongoing reduction in transistor sizes within the hardware. This trend allows for significantly more transistors per chip surface, potentially boosting chip performance and non-functional requirements like cost or energy efficiency. The continuously shrinking transistor substrate sizes and implicitly lower operating voltages [Com22] make them more susceptible to environmental effects [Shi+02; Con03; Bor05; Bau05; NX06; DW11]. Interfering impulses, aging, thermal effects, humidity, and radioactive and cosmic radiation, which can threaten the transistor's stability, are common hazards to transistors.

Overall, this trend makes hardware more vulnerable to permanent and *transient* hardware faults; *soft errors* is a term often used to refer to transient hardware faults. This situation introduces a new challenge in designing systems, known as the *"reliability wall"* [BJS07] or *"soft-error wall"* [Muk08]. Permanent hardware faults are consistently identifiable, whereas transient faults are momentary and vanish after a brief period. Consequently, these transient faults can alter the transistor's state, becoming untraceable.

For instance, if the transistor is part of a memory cell, this transistor-state change can lead to a *bit flip* in the cell, potentially resulting in an externally observable *system failure*. In response to these emerging threats, certification authorities require explicit measures to address transient faults in their safety standards [Sch16], such as IEC 61508 [IEC98] or ISO 26262 [ISO18].

Known system failures caused by transient faults are the unexpected pitch-down maneuver of a Qantas Flight 72 in 2008 [Bur08] and the uncontrollable acceleration of some Toyota Lexus ES 350 [Yos13; Koo14], for instance, which resulted in injuries or fatalities. Implementing protective countermeasures (*hardening*) to detect or rectify such bit flips is crucial to ensure the system's reliability or, at the very least, its safety. Cost optimization may have prevented implementing hardening techniques in the Lexus ES 350. [Yos13; Koo14].

A hardening technique should aim to detect or directly correct bit flips within the system. These bit flips should be corrected while the system is in operation. Implemented error *detection* and *correction* techniques protect the system from errors. Prominent examples of these techniques are *n-Modular Redundancy* [HHJ90; Yeh96; Sle+99; Rat04; Muk08], which replicate hardware or software n times, and a final result is determined using a voting mechanism, and *Error-Correcting Code (ECC)* [Muk08; Ros+11] implemented in software, which can correct bit flips in memory blocks during operation.

Hardening techniques can be implemented in hardware, offering high performance but limited flexibility and higher costs, mainly due to its obliviousness to application demands [Sch16]. In contrast, *Software-Implemented Hardware Fault Tolerance (SIHFT)* techniques are more flexible, cost-effective, and can be more finely tailored to specific purposes, making them popular [Rei+05b; Kon08; RGF23].

1.2 Research Context

To validate the effectiveness of hardening techniques, it is possible to actively flip bits in systems at specific times and bit locations during operation, ensuring the system continues running afterward. This process, known as *Fault Injection (FI)*, emulates transient faults [Cho+13; DC+14; Sch+15; Sch16]. The resulting externally observable behavior is classifiable based on whether the FI is *benign* or led to *unexpected* behavior, such as non-termination or outputting corrupted results.

Implementing or developing hardening techniques is challenging in practice. *Systematic* FI evaluates hardened systems' reliability and assesses hardening techniques' effectiveness. Such an evaluation involves systematically examining the resulting system behaviors when potentially transient faults occur. An *FI campaign* encompasses individual FIs executed independently to classify resulting behaviors at every possible point in time and bit location. The results of the FI campaign allow for the quantification of the overall system reliability using relevant metrics. This process and the obtained results allow iterative improvements to the system, measurably enhancing its reliability with each iteration.

The *Fault Space (FS)* [Gol+06] concept outlines all possible FIs of the *System Under Test (SUT)*. An FS $\mathscr{F}$ is the set of all time points T and bit locations L that the system handles during its operation: $\mathscr{F} = T \times L$. There are two approaches to handling the set $\mathscr{F}$ when assessing the reliability of the SUT, depending on the purpose of the analysis: (1) either FIs are probabilistically determined to cover the FS as completely as necessary, or (2) FIs are determined for each point in the FS to cover it completely, which is more time-consuming.

For a complete analysis of the SUT's reliability through FIs, the FI campaign must contain FIs whose resulting SUT behavior thoroughly covers the *whole* FS. Achieving complete coverage of the FS by executing an FI for every point in $\mathscr{F}$ is infeasible regarding the *exploding* execution time for such an FI campaign; the more points in the FS exist, the longer the execution of the SUT's program is. The campaign runtime can span several *Central Processing Unit (CPU)* years, even for smaller systems with high hardware performance, tiny memory size, and a very short program runtime. When seeking a complete statement on the outcome of an FI at any location and time of the SUT, the FS must be covered entirely, although using probabilistic approaches to FS's coverage can significantly reduce the campaign runtime.

However, two ways exist to reduce the overall FI-campaign runtime $t_{\text{cpn}} = n \cdot t_{\text{FI}}$: (1) The *pre-injection analysis* contains methods that reduce the number of FIs n needed for complete FS coverage before the campaign starts. (2) The second option is to *dynamically optimize* the execution time t_{FI} of individual FIs technically or by predicting resulting SUT behaviors post FI. Achieving full coverage of the FS sets an upper bound for the number of FIs computed in probabilistic methods. Thus, any method falling under (1) or (2) and ensuring complete FS coverage can be adapted for probabilistic approaches. Consequently, the primary focus of this work is on methods that achieve complete FS coverage.

As shown in Figure 1.1, an iterative analysis process cycle starts with the SUT. Using the methods in the pre-injection analysis (1) to accelerate an FI campaign impacts the number of executed FIs within the campaign. The FS $\mathscr{F}$ is also sketched in the figure, containing all possible FIs $T \times L$. The second option (2) occurs dynamically during the individual execution of the FIs within the FI campaign. The resulting behaviors from the campaign are used for the concluding reliability evaluation, leading to further improvements in SUT's hardening.

This iterative process is reiterated until the desired system reliability requirement is met.

Acceleration methods vary in two essential properties, depending on the context and whether methods reduce the FIs overall or dynamically accelerate single FIs:

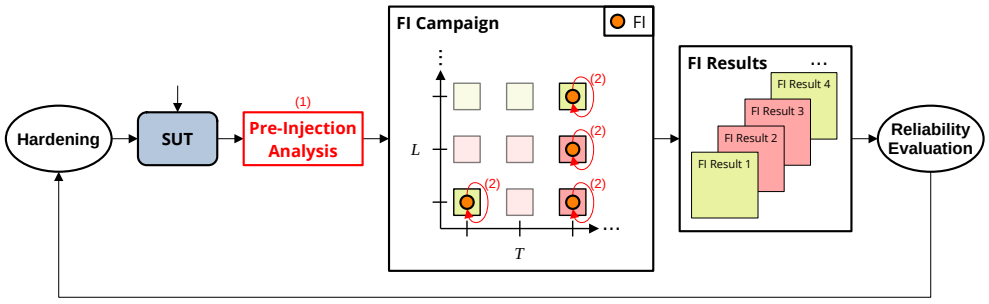


Figure 1.1 – Iterative Reliability Evaluation using Fault Injection.
The SUT starts with a pre-injection analysis (1) before executing any FIs, thus reducing the necessary number of FIs. Subsequently, the FI campaign executes individual FIs more swiftly through acceleration measures (2). The FIs from the initial step ensure that the FI campaign covers the entire FS $\mathscr{F} = T \times L$, defined by all possible points in time of the SUT execution T and all its bit locations L. The assessment of the SUT's reliability requires gathering all the FI results. Following the concluding evaluation regarding the current SUT's reliability, a subsequent version of the SUT receives new or improved hardening, which iteratively refines the SUT and allows it to undergo the same process as often as needed.

FS-completeness This property relates to the ability to make statements about the system behavior at specific points in the FS after the completion of the FI campaign. The scope here is the entire FS and is thus irrelevant to individual FI acceleration methods.

Precision An acceleration method is *precise* if it delivers deterministic, reproducible results without relying on heuristics; otherwise, it is considered *approximative*.

It is possible to combine acceleration methods, like combining the pre-injection analysis method *Def/Use Pruning (DUP)* [Smi+95; GS95; Ben+98b; BP03; Bar+05; Gri+12] and the dynamic acceleration method *checkpointing* [Par+00a; Par+00b; Ber+02; SH+14; Par+14], for instance. The well-established FS-complete and precise DUP use data read and data write access patterns to group FIs into sets of FIs with equivalent resulting SUT behaviors post-FI. Checkpointing restores recorded system states instead of executing the system entirely from the very beginning for every single FI. This results in a set or chain of acceleration methods that completely covers the FS and works precisely before and during the campaign's execution.

1.3 Purpose of this Dissertation

The chosen abstraction layer of the SUT influences the set of time points and locations for the FS. Selecting an abstraction layer for FI is a complex decision, with injectable layers ranging from the software application through the *Instruction-Set Architecture (ISA)* and gate logic down to the physics layer. Injecting closer to the physics ensures a more accurate overall estimation. However, low-level hardware models for FI may be unavailable or too complex to handle for practical FI [SB19]. High-level system layers, like the ISA layer, may be less precise but can be very fast [Cho+13] and ease understanding erroneous application behavior [Sch+15]. The ISA layer, as an interface between the software and hardware of a system, is well-suited for testing SIHFT techniques on a *fixed* hardware configuration and makes iterative improvements for systems with implemented SIHFT methods possible.

This dissertation focuses on accelerating FI campaigns on the ISA layer to evaluate a system's reliability on fixed hardware with implemented SIHFT techniques. Despite methods like DUP reducing the number of required FIs significantly by up to five orders of magnitude [Bar+05], the overall campaign runtime can still be huge, even when combined with other acceleration methods.

To further reduce the FI-campaign runtime and enhance its feasibility, I extract the *program structure* of the system's program. This extracted information is then used to reduce the campaign's runtime through specifically designed acceleration methods. Parnas defines program structure as the organization and arrangement of components, instructions, and data within a computer program, and it encompasses the overall architecture, control flow, data flow, and relationships between different parts of the program [Par72]. In this regard, I consider the *data flow*, the *control flow*, and their interweaving and influence on the program flow to reduce the number of FIs needed for whole FS coverage, irrespective of the resulting system behavior, and to accelerate single FI executions specifically.

The outcome of my research encompasses three contributions that reduce the execution time of FI campaigns on the ISA layer by utilizing extracted program structures. I distinguish between methods executed in the *pre-injection analysis* (refer to (1) in Figure 1.1) and methods that accelerate the execution of individual FIs, referred to as *dynamic fault-injection acceleration* (see (2) in Figure 1.1):

(1) Pre-Injection Analysis

Contribution 1 *Data-Flow–Sensitive Fault-Space Pruning*
The precise and FS-complete *Data-Flow Pruning (DFP)* method [▷PDL21] extends the well-established DUP by using information from the data flow, instruction semantics, and lifetimes of dynamic values, which works independently from the focused resulting system behavior. Specifically, the individual components of this contribution are:

- The concept of the DFP method for data-flow–sensitive ISA-layer FI reduction
- The utilization of the data flow, its instruction semantics, and value-dependent bit-wise local fault-equivalence rules
- Quantitative comparison against DUP with significant precise FI reduction

Contribution 2 *Program-Structure–Guided Approximation of Fault Spaces*
The approximative but FS-complete *Fault-Space Regions (FSRs)* concept [▷Pus+19], which uses dynamic jump addresses and data flows to reduce the number of FIs of DUP further. The individual components of this contribution are:

- The concept of forming FSRs for program-structure–guided ISA layer FI reduction
- I present two instantiations of the FSRs concept using basic blocks and function scopes
- Evaluation of this approach to demonstrate that this approximation reaches acceptable error rates compared to a precisely covering method, like the DUP

(2) Dynamic Fault-Injection Acceleration

Contribution 3 *Timeout-Detection Methods for Fault-Injection–Experiment Acceleration*
As part of this contribution, I explore non-terminating systems post-FI, also known as timeouts. My general findings on this topic are guidelines for developing timeout detectors for individual FIs. Following this, I describe my initial approaches that lead to the timeout detector ACTOR [▷Tho+22]. The components of this contribution are in detail as follows:

- A detailed analysis of the nature of timeouts, including an upper-bound analysis regarding the maximum achievable runtime savings, as well as the initial findings about my first approaches in predicting timeouts and the resulting takeaways for my concurrent work in this context
- The dynamic, heuristic acceleration method ACTOR uses autocorrelation [IKP16] based on dynamic jump addresses to detect whether a system terminates.
- Evaluation of ACTOR, its prediction-error rate, and the achieved FI-campaign–runtime end-to-end savings

1.4 Structure

This section describes the structure of this dissertation, gives an outline for every chapter, and assigns the three contributions to the respective chapters:

Chapter 2 *Dependable Computing and Fault Injection* (pp. 9-50)
This foundational chapter addresses essential elements of dependable computing and the potential impact of transient hardware faults. The chapter delves into the effects of transient hardware faults on systems and outlines strategies for hardening systems against such faults to enhance their reliability. Additionally, I elaborate on fault injection to test and gauge a system's reliability. Given that FI typically entails a substantial workload, I also explore the concept of FI acceleration methods aimed at optimizing the reliability evaluation process through FI.

Chapter 3 *Implementation and Evaluation Process* (pp. 51-70)
In this chapter, I introduce the used FI framework FAIL*, detailing its structure and the integration of my contributions into FAIL*, and explore the examined system failure classes. I establish the foundational concepts of effectiveness and efficiency in acceleration methods, which are essential for future evaluations, and establish the ground truths in this dissertation. Furthermore, I provide insights into the setup of my dissertation, encompassing my benchmark portfolio and the technical setup used across all evaluations in this work.

Chapter 4 *Data-Flow–Sensitive Fault-Space Pruning* (pp. 71-98)
Here, I delve into my first contribution, outlining the *Data-Flow Pruning (DFP)* methodology [▷PDL21]. Leveraging the extracted data flow from the executed program of the SUT and the inherent instructional semantics, I utilize masking effects to minimize the required number of FI. I comprehensively describe the DFP in this chapter, showcasing its effectiveness and efficiency in reducing the number of FIs.

Chapter 5 *Program-Structure–Guided Approximation of Fault Spaces* (pp. 99-120)
In this chapter, I introduce my second contribution, the *Fault-Space Regions (FSRs)* [▷Pus+19], established through extracted program structures, specifically the dynamic jump targets of the program. The FSRs define the temporal segmentation of the FS, offering a method to diminish the necessary number of FI. I describe this approach in detail within this chapter and present the subsequent FSRs' evaluation results.

Chapter 6 *Timeout-Detection Methods for Fault-Injection–Experiment Acceleration* (pp. 121-144)
In contrast to my previous contributions to reducing the necessary FIs, this chapter delves into accelerating the execution of individual FIs. Here, the primary focus is timeouts, which signify potentially non-terminating systems after completing an FI. In this chapter, as the third

contribution, I extensively explore timeouts within the context of FI. Subsequently, I describe the methodologies I have developed to detect timeouts at an early stage, ultimately leading to the creation of the timeout detector ACTOR [▷Tho+22]. Finally, I detail the functionality of ACTOR and present its evaluation results.

Chapter 7 *Discussion and Related Work* (pp. 145-162)
In this part of my dissertation, I delve into essential facets of ongoing FI-related research, bridging key concepts with my contributions. I discuss fundamental aspects within the FI-accelerating research landscape and this work. Concluding, I present related research pertinent to my contributions.

Chapter 8 *Conclusion* (pp. 163-167)
I aim to recapitulate my dissertation's core topics and contributions in this final chapter. I revisit the key elements and potential future-work topics that emerged during my research.

1.5 Typographical Conventions

Throughout this dissertation, I use the *italic* font shape to emphasize an essential term or fragment of a sentence. When presenting figures from previously published articles, I use the phrase "*The figure is adapted from [citation]*" to specify a modified or recreated version from the source, and "*The figure is from [citation]*" to denote an unaltered one. In my citation marks, an open triangle denotes that I am the primary author or one of the co-authors (e.g., [▷PDL21]). I typeset source code and in-text references to source-code identifiers (like `function()`, `variable`) in a monospace font.

2

Dependable Computing and Fault Injection

> Don't only practice your art, but force your way into its secrets, for it and knowledge can raise men to the divine.
>
> LUDWIG VAN BEETHOVEN (1770–1827)

This elementary chapter provides a basis by explaining key concepts for understanding the subsequent topics. It begins with exploring dependable computing *terminology* and dependability's significance in ensuring computer systems' reliability and performance.

The chapter overviews *transient hardware faults* and their potential impact on computing systems and sketches countermeasure approaches to make a system less prone to these transient faults.

It dedicates *fault injection* as a testing technique. This technique involves explicitly injecting faults into a system to assess its reaction, identify vulnerabilities, and evaluate the effectiveness of implemented countermeasures.

The chapter's final section introduces the dissertation's core topic – *reducing* end-to-end runtimes in fault-injection campaigns comprising systematic fault injections. This reduction is pivotal for conducting efficient and practical assessments of system reliability.

2.1 Dependable Computing

In the world of safety-critical systems, a precise definition of terms is inevitable, and Figure 2.1 illustrates the most essential terminologies as the *dependability tree*.

The first aspect of dependability is the differentiation of five different *attributes*. These attributes describe related concepts but address different aspects of dependability. The differentiation and delimitation of different *threats*, in terms of faults, errors, and failures, is also essential to distinguish between their causes and effects.

We will learn about the increasing *demand* for transistors and the trends regarding the *size* of transistors in the future. Afterward, I will present the physical causes of *transient hardware faults* and how shrinking transistor sizes intensify the probability of transient hardware faults.

A *fault* triggers the *fault propagation chain*, activating an *error*, and further propagation can lead to *failures*. We will also look at the assessment of dependability and the *means* to enhance dependability in four different ways. I will present some common error-detection and error-correction methods to help understand how these concepts work. The primary focus of this dissertation is *fault forecasting*, an essential concept in dependable computing that estimates erroneous system behavior during the occurrence of a transient effect.

For the remainder of the dissertation, I will use the term *system* to refer to a system with the intended, fixed hardware configuration and the corresponding software in the target environment.

2.1.1 Attributes of Dependability

The system's behavior results from implementing the functionality outlined in a specification. When the system's behavior deviates from the specification, it fails to fulfill its intended task. The impact of this divergence can vary significantly depending on the domain or use case. In safety-critical environments, minimizing or even preventing any imminent danger is paramount. A system is *dependable* when it can effectively manage impending risks, maintaining an acceptable level of risk while ideally adhering to its specification or, at the very least, not violating it beyond a defined degree.

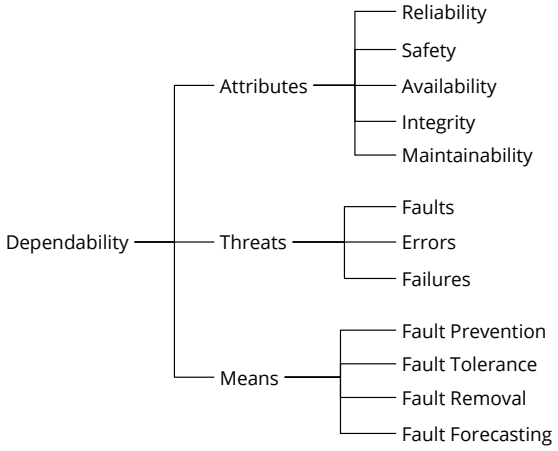


Figure 2.1 – Dependability Tree.
Dependability and its attributes, threats, and means (based on [Avi+04])

A system's *dependability* is defined by its following five attributes [Avi+04]. Figure 2.2 shows an overview of these five attributes and their relations.

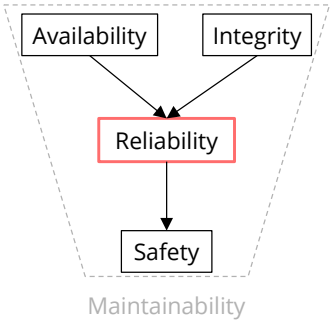


Figure 2.2 – Relations of the Dependability Attributes.
This figure consists of the five *dependability* attributes and their relations to each other. High *availability* and *integrity* are prerequisites for a system to be *reliable* because only then it can perform its task. *Safety* depends on reliability. If a system is not reliable, it cannot be safe. It may only be safe if it is reliable if it prevents misbehavior. *Maintainability* affects all the other attributes.
This work focuses mainly on the attribute *reliability*.

Availability This attribute is *"that a system is functioning at a particular instant of time"* [Muk08]. The system operates continuously, except for planned downtime for maintenance or unexpected system outages. In an ideal scenario, the system is available 100 percent of the time and can initiate its designated tasks. Availability indicates whether the system is operational without guaranteeing it operates without failure.

Integrity The meaning of integrity in the context of dependability is the *"absence of improper system alternations"* [Avi+04]. It pertains to the components of a system that are essential for its infrastructure or data structures. A system has high integrity when it can be assured, under all circumstances, that it relies on the logical correctness and completeness of both hardware and software, as well as any valid data (regardless of the data's correctness). This also implies that the system should consistently maintain valid system states, regardless of whether it fulfills its specified function.

Reliability Reliability is often used interchangeably with *robustness* or *functional correctness* in the research community. Reliability stands for the *"continuity of correct service"* [Avi+04] during the execution of the system. Reliability is a crucial aspect of dependability, but it depends on the prior conditions of availability and integrity. Reliability refers to the system's capability to execute its specified function within a defined response time. In this context, what matters is the system's observable behavior, regardless of any undesired internal system states. The sole criterion for reliability is that the system's behavior remains correct even in the event of an internal error or a loss in integrity. Integrity and reliability are interrelated because a system with lower integrity might not be reliable as it fails to fulfill its tasks. However, a system can terminate in a valid system state (high integrity) but not terminate correctly due to an internal error. If such a case proceeds undetected, and the system is considered to be operating correctly, it is termed a *Silent Data Corruption (SDC)*.

Safety With safety, Avizienis means the *"absence of catastrophic consequences"* [Avi+04]. When a system becomes unreliable or an intented task is expected to fail in any way, it is *safe* if it can prevent catastrophic consequences. In specific real-time scenarios, a system may be shut down or restarted to bypass an error and avoid potentially dangerous outcomes. However, even if a system is deemed reliable, external environmental factors can still impact its safety. In the automotive context, for instance, the pedestrian protection system [GT07] is designed to minimize the risk of pedestrian injury, prioritizing safety in such scenarios.

Maintainability This attribute is essential to ensure the system's design allows for maintenance, modifications, and repairs, indirectly supporting the other four attributes.

As Figure 2.2 shows, these five attributes are interrelated but address different depenability aspects. In this dissertation, I focus on attribute *reliability*, which influences *safety* and depends on *availability* and *integrity*. This dissertation does not consider *maintainability*.

2.1.2 Growing Danger to Hardware Reliability

This work delves into reliability concerning the correct execution of program code on the hardware. The assumption of correct execution is fundamental in software development, relying on the proper functioning of the hardware, specifically the *transistors* that comprise it in hardware.

The demand for transistors has increased since the late 70s and continues its exponential growth [Com22], driven by the increasing digitalization of everyday life for end-users and the ongoing reduction in transistor sizes. The continuous reduction in transistor size allows for more transistors per single chip. Gordon Moore defined this development in a reduction in transistor sizes as *Moore's Law* in 1975, which describes that the number of transistors on the same chip surface approximately doubles every two years[1] [Sch97], a trend that persists today.

The increasing number of transistors on a single chip surface leads to enhanced hardware performance, but it also introduces new challenges, such as the *memory wall* and *power wall* [BJS07; Muk08]. The memory wall has slowed overall performance development in computer systems because powerful processors outpaced memory speeds, causing memory to become a bottleneck. Additionally, the densification of microstructures results in condensed particle charges, leading to increased power dissipation and overheating issues, known as the power wall.

These walls have varying impacts on hardware development; various architectural solutions have been designed to partially mitigate these challenges, including techniques like prefetching and multi-threading and reducing supply voltages and clock frequencies [Muk08; Mit14; Com22]. However, these solutions often come at the expense of single-thread performance, which does not keep pace with the rapid increase in transistors [Com22].

Another increasingly relevant challenge is the *soft-error wall* [Muk08] or *reliability wall* [BJS07]. The continuous reduction in transistor sizes decreases the charge quantity within transistors, increasing the likelihood of erroneous behavior. This requires lower operating voltages, which can be advantageous for addressing the power wall but also makes the hardware more susceptible to errors. Although additional hardware can compensate for potential incorrect behavior to some extent, it consumes more energy. Implementing is not always straightforward due to constraints like chip size, power usage, and performance limitations [Sch16].

[1] Moore's Law is not a physical law but a general guideline. Originally proposed in 1965, it initially stated that the number of transistors on a chip would double yearly. However, this prediction was revised to a two-year interval a decade later in 1975.

Another crucial factor impacting reliability is the hardware's life cycle. This life cycle is often described as the *bathtub curve* [Wil02a; Wil02b; LB17] and consists of three distinct phases, as shown in Figure 2.3:

Infant Mortality This initial phase where manufacturing defects occur during development, even before the hardware reaches the market. Manufacturers aim to identify and rectify these defects to enhance hardware quality. This phase incurs a significant portion of production and development costs [Hof16], making it a potential area where time and costs could be saved at the expense of reliability.

Normal Life In this phase, the hardware operates as intended, with a relatively constant probability of erroneous behavior over time [Wil02b; LB17]. Transient hardware faults resulting from internal and external interference sources cause around 80 to 90 percent of errors during this phase [CS90; Lal97].

Aging Phase In the final phase of the life cycle, aging processes significantly increase the probability of erroneous behavior, which can lead to permanent, irreparable defects in the hardware caused by *time-dependent dielectric breakdown* [McP07]. Decreasing transistor sizes also significantly increases the susceptibility to these breakdowns and thus generally shortens the operational lifetime of the hardware until it becomes unusable [Sri+04].

Hence, we have identified two significant factors that threatens hardware's reliability: (1) Moore's law, which suggests that transistors continually shrink in size, brings about persistent challenges in hardware development. (2) Notably, the reliability wall becomes a significant obstacle to the progression of technology. This wall's impact is most pronounced during the normal-life phase (see Figure 2.3) of hardware, primarily due to transient hardware faults, which is the main reason for erroneous behavior [CS90; Lal97]. Furthermore, the constant reduction in transistor size amplifies this issue. In summary, as transistors become smaller and smaller, the likelihood of transient hardware faults increases, leading to a higher reliability wall.

The combination of Moore's law and the reliability wall presents a complex challenge for the dependable operation of modern hardware systems.

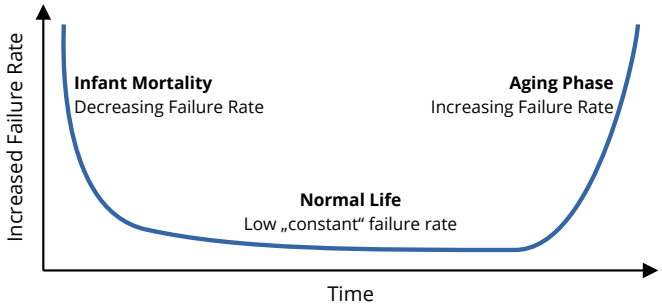


Figure 2.3 – The Bathtub Curve as the Life Cycle of Hardware.
This illustration depicts the failure rate of hardware over time, delineated into three distinct phases. In the *Infant Mortality* phase, the failure rate is initially relatively high. As the hardware undergoes iterative development and transitions into productive use, it enters the *Normal Life* phase, characterized by a low and relatively constant failure rate. In the final phase, known as the *Aging Phase*, the failure rate experiences a significant upsurge due to the effects of aging. *The figure is adapted from [Wil02a].*

For instance, in a 6T-*Static Random-Access Memory (SRAM)* cell, a charged particle can trigger a transient fault in one of the six transistors, potentially causing the SRAM cell to change its state or access type. More specifically, due to transient transistor faults, the stored bit within the SRAM cell can be *inverted* or *flipped* from a *logical zero* to a *logical one* or vice versa [Zie+96]. This is known as a *bit flip* resulting from a transient fault, also referred to as a *soft error* or *Single-Event Upset (SEU)*[2] [GWA79].

2.1.2.1 Transistor Trends

As early as 1978, the Intel Corporation observed transient effects induced by radioactive radiation from contaminated casing materials [MW79]. In the same year, cosmic rays were identified as another possible cause of transient semiconductor faults [ZL79]. The initial concept of the critical charge Q_{crit} for transient faults in semiconductors [MW79] remains relevant in semiconductor manufacturing today [Com22].

Due to the continuous miniaturization of transistors, the critical charge Q_{crit} has been decreasing, making cosmic rays an increasingly important factor in aeronautical engineering. Cosmic rays have become highly relevant in the upper layers of the atmosphere [TN93]. Normand and colleagues [Nor96a], however, discovered in 1996 through experiments that cosmic rays can cause operational failures even at sea level, which IBM further confirmed by many years of studies, which observed the influence of cosmic rays even in standard computer systems [Zie+96] and through observations of failures in Sun UltraSPARC-II workstations, which were also attributed to radiation-induced bit flips in *Static Random-Access Memory (SRAM)* cells [Hof16]. Subsequent research increasingly highlighted single-bit errors in SRAM cells and vulnerabilities in *Dynamic Random-Access Memory (DRAM)* cells concerning transient faults [Sri+15].

In 2001, semiconductor manufacturers expressed concerns as transistor structure widths decreased to less than 100 nm, suggesting that transient faults could affect not only memory cells but also logic elements [All+02]. This statement began understanding the *"reliability wall"*, essential in today's hardware manufacturing.

New transistor technologies, such as *FinFET*, *LGAA*, *LGAA-3D*, and *VGAA* transistors, occurred to overcome this wall and enhance chip performance. Currently, technologies with feature sizes as small as 7 nm are on the market, with developments progressing towards 3 nm. The International Roadmap for Devices and Systems committee predicts that these technologies will reach feature sizes of less than 1 nm by 2034 [Com22].

In line with this trend, transistors are operated with ever-decreasing operating currents [Hoe12; CS17; Com22], while their substrates continue shrinking and their charge decreases [Hoe12; Com22] . This exacerbates the challenge posed by the reliability wall. To illustrate the concept of lower charge, consider the *channel length* l as a standard metric for transistor sizes. Figure 2.E2 shows the distance between the two semiconducting materials connected to the source and drain terminals. In the case of *Metal-Oxide–Semiconductor Field-Effect Transistor (MOSFET)*, a shorter channel length means fewer silicon ions form the channel. With a covalent radius of approximately $102\,\text{pm} = 0.102\,\text{nm}$ [Bon64], a channel with a length of around 1 nm contains only about ten silicon ions. Consequently, the smaller the channel, the more susceptible it is to disturbances and charge changes. The current trend is pushing transistors and their functionality to their physical limits.

[2] To provide more precise terminology, the case described here corresponds to a *Single-Bit Upset (SBU)*, as it involves explicitly a single bit flip. In addition to SBU, there are other categories, such as *Multiple-Bit Upset (MBU)* or its particular case, *Multiple-Cell Upset (MCU)*, where multiple states or bits are changed simultaneously by a transient effect. The term SEU generalizes all these events, as per references [Nic10; Cor13]. Throughout this dissertation, the term SEU is used interchangeably with SBU. MCU and MBUs are not extensively covered in this dissertation, but I will delve into MBUs again in Section 2.2.4 and Section 7.1.1.

2.1.2.2 Observed Occurrences of Transient Hardware Faults

The current trend significantly impacts memory cells or even logic circuits in the form of an SEU, which can propagate to higher abstraction layers and manifest as *externally observable* system failures or unexpected system behavior. As transient hardware faults do not result in permanent system damage, it can be challenging to identify whether a transient effect has occurred, especially in the context of increasingly complex hardware systems. In the following, I will provide examples of unexpected system behaviors that were externally observable. However, it is not always possible to conclusively attribute these issues to transient faults:

- In 2003, electronic voting was implemented in the Belgian elections. However, during the analysis of the election results, a peculiar anomaly emerged: one candidate received more votes than the total number of eligible voters in the corresponding electoral district. Subsequent investigations sought to uncover the root cause of this discrepancy. The candidate had received exactly 4096 more votes, which is a power of two and strongly suggests a bit flip in the electronic voting system [Gla04; Pau04].

- Server systems, like other critical systems, are subject to the five dependability attributes discussed in Section 2.1.1. In 2003 and 2004, a series of significant failures in server systems were documented. These failures had a severe impact on the system's functionality. The root cause of these failures was SEUs, which compromised the systems' ability to ensure reliability and availability [Mic+05].

- The incident involving Qantas Flight 72 in October 2008 resulted in injuries caused by a bit flip. The air data inertial reference units processed data that an SEU had corrupted. This corruption led to an unintended pitch-down maneuver initiated by the aircraft's autopilot, resulting in a descent with a force of -0.8 times the Earth's gravity. Subsequently, the aircraft executed a maneuver with a force of 0.2 times Earth's gravity, causing serious injuries to the passengers [Bur08].

- In 2008, Infineon highlighted the significance of transient effects on automotive systems and discussed potential countermeasures [Kon08]. Just a year later, the impact of these effects became a reality. Reports surfaced of drivers experiencing uncontrollable acceleration in Toyota Lexus ES 350 vehicles. The initial assumption was that the issue was related to the accelerator pedal being stuck due to a floor mat. However, further investigation revealed that SEUs were the primary cause of this behavior. The problem stemmed from the electronic acceleration system's vulnerability to bit flips, leading to accidents that resulted in various fatalities [Yos13; Koo14].

- In the context of digital certification, a *Certification Authority (CA)* employs Merkle trees to verify the validity of certificates. When a new entry appends the tree, a new leaf in the tree contains a new hash based on the hashes of all its predecessors up to the root. A posted report, *"A cosmic ray just murdered a Certificate Transparency log"* [Val21], brought attention to a potential transient fault that resulted in a security vulnerability. In the case of the Merkle tree shard Yeti 2022 by the CA company Digicert, the certificate validation process no longer worked correctly when checking from a leaf upwards. Finally, this issue was attributed to a possible bit flip in the hash of a leaf [JR21].

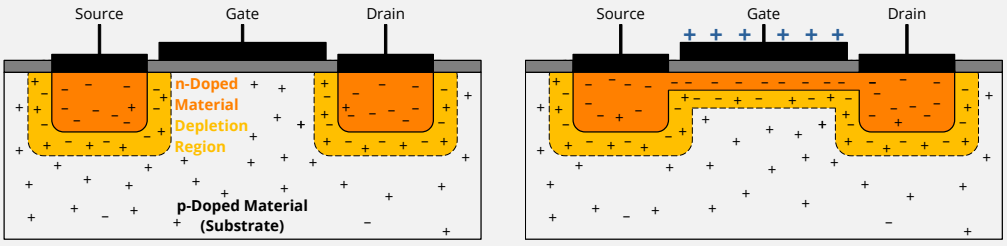
- In the gaming world, gamers often engage in speedruns, where they compete to complete a computer game as quickly as possible, sometimes for prize money. In a 2013 speedrun of

Super Mario 64, a speedrunner unexpectedly warped up several layers in the game, giving them an unexpected advantage. This phenomenon remained a mystery for years until it was finally determined that an SEU must have caused it, which is consistent with a bit flip affecting the variable that tracks the height of the in-game character [Bur20; Ram21].

As the list above shows, transient effects that are externally observable exist in various domains. Although they may be entertaining in the context of a computer game, they can pose significant risks in matters related to safety or security. It is crucial to consider and address transient effects in transistors to ensure the reliability of systems.

EXCURSUS: Transient Faults in Metal-Oxide–Semiconductor Field-Effect Transistors

To understand how *transient hardware faults* occur, a basic understanding of the functionality of transistors is necessary.
The most common and widely used type of transistor is the *Metal-Oxide–Semiconductor Field-Effect Transistor (MOSFET)*. One common variant of the MOSFET is the *n-Channel Enhancement Mode MOSFET (nMOS)*, and Figure 2.E1 illustrates its functionality. The MOSFET features three terminals: *source*, *gate*, and *drain*. Technically, a MOSFET has four terminals. The fourth terminal the *body* or *bulk*. It is usually internally connected to the source terminal to prevent unintended additional electric fields and voltage shifts at the transistor's boundaries. This detail is not crucial for a basic understanding of transistor functionality, but those interested in a more in-depth exploration can refer to literature like Liening [LB17] for more information.



(a) Non Conducting **(b)** Conducting

Figure 2.E1 – Functionality of an n-Channel Enhancement Mode MOSFET.
An enhancement-mode MOSFET features three terminals that enable it to transition between conducting and non-conducting states. A positive voltage is applied to the gate terminal to enable the flow of electrons between the source and drain terminals. This voltage shift induces a change in the charges within the n-doped and p-doped materials of the MOSFET, leading to the formation of a conducting n-channel between the source and drain terminals.

The source and drain terminals are connected to semiconducting material. This material is produced through a process known as *doping*. In the case of an nMOS, the source and drain materials consist mainly of negatively charged particles, typically electrons. As a result, they are called *n-doped* materials [CH02; WH11]. *p-doped* material known as the transistor *substrate* separate these n-doped materials. The transistor substrate can hold more positive charges [CH02; WH11]. The gate terminal is positioned between the two n-doped regions

and connects an electrically insulated layer, usually made of silicon dioxide. This insulating layer separates the gate terminal from all the semiconducting materials.
A voltage between the source and drain creates a *depletion region* at the transition between the n-doped and p-doped materials. Both of these regions function as diodes, preventing the flow of electrons from source to drain and vice versa. In contrast to the *enhancement mode* described here, the opposite *depletion mode* initially establishes a conducting channel between the diodes instead of preventing the flow.
Figure 2.4a shows the non-operating or non-conducting state of an nMOS. The gate terminal is crucial to enable the flow of electrons between the source and drain. When a positive voltage applies to the gate terminal, it shifts charges within the substrate. Initially, a depletion region forms beneath the gate's insulating layer, connecting and uniting the existing regions. When the voltage surpasses a certain threshold, it causes the electrons beneath the insulating layer to create a conducting channel between the source and drain, as illustrated in Figure 2.4b. This phenomenon is known as the *field effect* [WH11; DHE12]. The channel, which carries electrons, is referred to as an *n-channel* due to the negative charge of the electrons.
In addition to the *npn* arrangement of semiconducting materials in an nMOS transistor, a *pnp* arrangement is possible, resulting in a *p-Channel Enhancement Mode MOSFET (pMOS)* transistor. Consequently, the formed *p-channel* transports positive charges when a negative voltage is applied to the gate terminal.
The backbone of the most commonly used CMOS technology for manufacturing processors and SRAM cells, consists of nMOS and pMOS transistors. They are applied complementary, where nMOS circuits represent the desired logic, and complementary circuits with pMOS transistors minimize noise and reduce power consumption. For example, a *6T-SRAM* cell is composed of six MOSFETs. This cell stores a bit using four MOSFETs (two nMOSs and two pMOS) that form two cross-coupled inverters. Two additional nMOS transistors control read and write accesses. Even though DRAM cells use capacitors to store a bit, nMOS transistors are still employed for accessing the cell [JWN10; WH11].
Transistors, essential components of processor circuits and power cells, must ensure proper functioning to guarantee that the hardware constructed with them reliably performs its intended tasks.
In addition to the aging processes, transistors are vulnerable to internal and external factors that can result in hardware failures during their "normal life" phase, as depicted in Figure 2.3. These failures are not permanent and do not make the hardware unusable; instead, they involve unpredictable and transient effects that can *temporarily* alter the state of a transistor, potentially impacting the hardware's behavior.
Transient faults can arise from various sources. Internal sources of transient faults may include crosstalk from adjacent interconnects or signal reflections at interconnect ends, leading to noise or signal delays [Con03; CMV02; RCE02].
On the other hand, external sources of transient faults can be fluctuations in the supply voltage, electromagnetic interference, and, most notably, *ionizing radiation*. Both *particle radiation*, such as alpha, beta, and cosmic high-energy neutron radiation, and electromagnetic radiation in the form of X-rays or gamma radiation have the potential to induce a charge shift within the transistor, which, in turn, can invert the state of a logic circuit [Zie+96; Muk08; Hof16].
Figure 2.E2 schematically illustrates the ionization force when a particle interacts with a transistor. During the particle's path through the transistor material, the deceleration of the

particle leads to the creation of electron-hole pairs within the material. This, in turn, results in an expanded depletion region, causing an *ion shift* that generates a transient conducting channel or a power pulse.

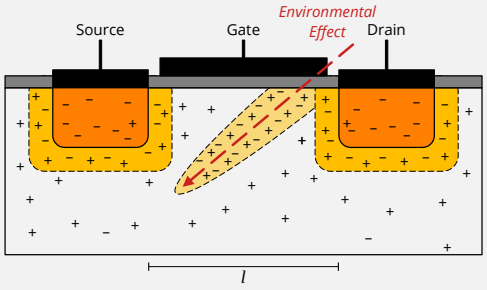


Figure 2.E2 – Influence of an Environmental Effect in a MOSFET.
This figure depicts the non-conducting state of a MOSFET, as shown in Figure 2.4a. An external environmental effect induces an arbitrary transient conduction channel through an ion shift. When the emerging depletion region expands to connect the two diodes sufficiently, it transiently functions as a conduction channel between the source and drain terminals despite the original non-conducting state of the transistor.

In the case of a transistor in a conducting state, this ion shift disrupts the existing conducting channel between the diodes [Bau05]. Consequently, it induces a short-term, transient, and undesired state of charge within the transistor. If the resulting charge, which depends on the initial kinetic energy of the particle, surpasses a *critical threshold* Q_{crit}, the state of the transistor switches [MW79].

2.1.3 The Fault Propagation Chain - Fault, Error, Failure

After discussing the potential causes of transient hardware faults and their underlying physical reasons, I will delve into the specific consequences of these faults or *threats*, as mentioned in the dependability tree (refer to Figure 2.1), on the behavior of a software-executing hardware system. Establishing a precise definition of what constitutes a *system failure* is essential. Throughout this dissertation, when referring to a *system*, I am generalizing it to encompass any hardware that executes software designed to fulfill a task as specified during the design or requirement process.

As previously discussed, every system aiming for reliability faces the threat of faults. These faults can manifest as errors, ultimately leading to system failures. A linear *fault propagation chain* visualize this progression: fault → error → failure → fault → error → failure → ⋯ [Lap+90; Avi+04]. A transient hardware fault can serve as the initial trigger for such a chain.

Exceeding the critical charge in a transistor results in a change of its state, referred to as a *fault*. However, a faulty charge state in a transistor does not necessarily lead to unexpected system behavior or misoperation. If the affected transistor is recharged after the change or remains unused during a system's execution on the hardware, the fault is considered *passive* or *dormant* [Lap92; ALR01; Avi+04; Gol+06]. In this context, a passive fault does not impact the correct execution of a program, even if a faulty charge state exists within the system.

The fault is *activated* once the system uses a transistor with a faulty charge state and transforms into an *error*. At this point, the fault impacts the system state and becomes an *effective fault*. However, an error does not necessarily result in a system misbehavior and can be *masked* by side effects. An

example of error masking is an AND gate connected to an error-holding memory cell. If any other input of the AND gate carries a *zero*, the error is masked and does not influence the system.

Once a fault is activated, an error exists in the system, which may *propagate* from one layer to another. If the error eventually reaches the system's boundary and results in externally observable misbehavior, it is a *system failure*. A system failure indicates that the system no longer serves its intended purpose.

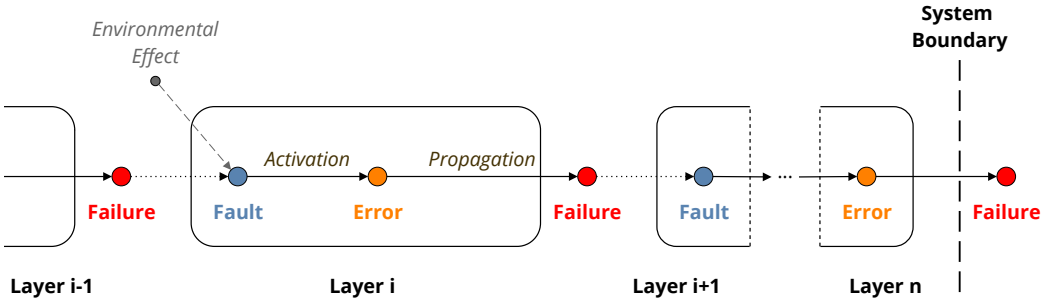


Figure 2.4 – Fault Propagation Chain.
A fault, whether caused by environmental effects or the failure of the previous layer, has the potential to be activated and transformed into an error. This error can then propagate to the layer boundary, resulting in a layer failure. The layer failure, in turn, becomes a fault in the subsequent layer. If this sequence continues across multiple layers and ultimately reaches the system boundary, it culminates in an externally observable system failure.

Figure 2.4 illustrates the interconnection of individual chain links across multiple layers. Consider a system comprising n layers. *Environmental effects* can induce a *fault* at layer i, initiating the *fault propagation chain*. If this fault is *activated*, it results in an *error* at layer i. When this error subsequently *propagates* to the boundary of layer i, a "layer failure" occurs, causing a fault at the next layer, layer $i+1$. This process can propagate up to the final layer n. Suppose an error on layer n propagates to the *system boundary*. In that case, it leads to *externally observable* unexpected system behavior, resulting in a *system failure* where the system no longer fulfills its specified tasks.

Consider various *system layers* and how the fault propagation chain can affect them, as depicted in Figure 2.5. The fault propagation chain has the potential to transmit a fault caused by an SEU at the physical layer across all abstraction layers of a system. An SEU, induced by environmental influences, results in an undesirable charge state of a transistor on the physical layer. This fault can lead to a malfunction on the physical layer, manifesting as a fault on the digital-logic layer.

The propagation of these faults can continue up to the application layer, ultimately causing the original fault on the physical layer to result in externally observable misbehavior. Therefore, if a fault on the physical layer is effective and the subsequent propagation through the layers extends beyond the system boundary, it leads to a system failure.

In the context of an automotive controller designed to control an *Anti-Lock Breaking System (ABS)* in a car, the propagation of a fault through the layers up to the system failure can be outlined as follows:

Physical Layer A transient effect induces a transistor to switch.

Digital-Logic Layer The affected transistor is part of an OR gate on this layer. One input is a logical *zero*, and the other is a logical *one*. The transistor's switch in the physical layer causes both inputs to become *zeros*, leading to *zero* output.

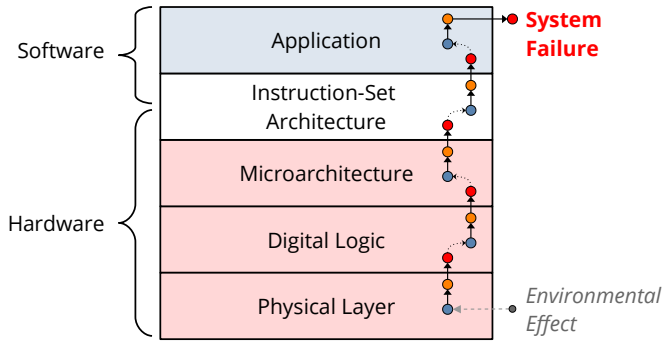


Figure 2.5 – Fault Propagation Chain in Exemplary System Layers.
An SEU within a transistor triggers a fault at the physical layer due to environmental effects. Once this fault is activated and the resulting error propagates, it results in a physical-layer failure. This layer failure, in turn, induces a fault at the next layer, in this case, the digital-logic layer. This sequence can continue upward to the application layer, eventually resulting in an externally observable system failure.

Microarchitecture Layer The `OR` gate is part of a half adder within the *Arithmetic Logical Unit (ALU)* on this layer. The ALU performs the binary addition $(\texttt{1001 0001})_2 + 1_2 = \texttt{0x92}$. The change in the `OR` gate's output alters the first bit in the first summand from $(\texttt{1001 0001})_2$ to $(\texttt{0001 0001})_2$, resulting in the incorrect result `0x12`.

***Instruction-Set Architecture (ISA)* Layer** The instruction `ADD r1, #1` is in progress. The incorrect calculation from the previous layer leads to an erroneous result, `0x12`, stored in register `r1`, which differs by one bit from the correct result.

Application Layer In our example application, which directly controls the car's ABS, the incorrect value from the previous layer causes a pointer variable to change. This variable should dereference the next byte at memory address `0x92` in periodically updated sensor data to trigger the ABS. However, after the fault, the variable points to address `0x12` instead, leading to a pointer dereferencing the wrong byte.

This example highlights the propagation of a fault and emphasizes that it is not crucial on which layer the fault propagation chain breaks to prevent a system failure. Once a fault is caught on one layer, it does not propagate to the next, illustrating the importance of fault- and error tolerance-mechanisms at various system layers. I will delve into this topic in more detail in Section 2.1.5.

2.1.4 Dependability Metrics

In earlier sections, we delved into the causes and potential consequences of faults in a system. Dependability metrics allow the *comparison* and *quantification* of these effects, . These metrics find utility in various contexts, such as system design, where they enhance dependability through iterative improvements. The collection of these individual metrics listed here is based on the collection from Schirmeier's dissertation [Sch16].

The most fundamental metric, *Time To Failure (TTF)*, quantifies error rates, although it pertains explicitly to failures[3] [Muk08]. If a system experiences an error after x years of operation, this duration is referred to as its TTF.

In a more practical context, *Mean Time To Failure (MTTF)* represents the mean time between multiple errors within the same device or the mean time of TTF for multiple devices collectively [Muk08].

Closely related to MTTF, *Mean Time To Repair (MTTR)* measures the mean time required for inherent device repair (I will revisit this in Section 2.1.5). Additionally, *Mean Time Between Failures (MTBF)* denotes the mean time between multiple errors, encompassing the time needed for repairs: MTBF = MTTF + MTTR.

With MTTF, one of the most widely known and used metrics for systems is *Failure In Time (FIT)* [Muk08; Ass21], where 1FIT *"represents an error in a billion (10^9) hours"* [Muk08].

An advantage of FIT over MTTF is that FIT is additive. Therefore, the FIT of the entire system, denoted as $\text{FIT}_{\text{system}}$, is estimated by adding the FITs of the system's components [**marwedel:21:book**; Muk08]:

$$\text{FIT}_{\text{system}} = \sum_{i=1}^{n} \text{FIT}_{\text{component } i}$$

FIT and MTTF are inversely related [**marwedel:21:book**; KK07; Muk08]:

$$\text{MTTF} = \frac{10^9\,\text{h}}{\text{FIT}}$$

Mukherjee introduced a specific *reliability metric*, denoted as $r(t)$. This metric represents *"the probability that the system does not experience a user-visible error*[4] *in the time interval* $(0, t]$*"* [Muk08]. In simpler terms, $r(t)$ quantifies the probability p that a system continues to function correctly and fulfill its intended purpose until time t, without failing.

When considering a total of n similar systems at time t, and $f(t)$ of them have failed by that time, the reliability $r(t)$ can be calculated as follows:

$$r(t) = 1 - p = 1 - \frac{f(t)}{n}$$

In the case of systems with a constant and known failure rate λ, the reliability metric follows an exponential function:

$$r(t) = e^{-\lambda t}$$

The exponential relationship between reliability and time is referred to as the *exponential failure law* and is commonly used in transient fault analysis, as noted by Mukherjee [Muk08]. The expectation of $r(t)$ corresponds to the MTTF and is equivalent to λ.

Once this dissertation has addressed some other fundamentals, it will discuss of additional metrics to measure dependability in more detail. I revisit the discussion of additional metrics to measure dependability in more detail later once some other fundamentals have been addressed in this dissertation.[5]

[3] The research community frequently uses the terms *fault*, *error*, and *failures* almost interchangeably, which is not entirely accurate because these terms refer to different effects within the fault propagation chain [Lap+90; Avi+04]. Whenever a metric *name* mentioned here includes the words "fault" or "error", it often refers to a failure. Conversely, the failure mentioned in the TTF metric in Mukherjee's description [Muk08] pertains to an error in the context of the fault propagation chain, which, unfortunately, is also a common terminology issue within the research community. There is no uniform standard, establishing and explicitly defining these terms is highly advantageous, as done in Section 2.1.3.

[4] To clarify further, when Mukherjee refers to a failure, he is specifically addressing a failure within the context of the fault propagation chain (as discussed in Section 2.1.3).

[5] I will explore other well-known dependability metrics in Section 2.2.2.

2.1.5 Enhance the Dependability

Eliminating the sources and causes of faults and errors is essential to achieving a dependable system free from failures and disrupting the fault propagation chain. Since environmental effects that lead to faults are non-deterministic and random, it may not always be possible to eradicate their underlying causes. However, it is possible to reduce or identify faults while they are occurring. This optimization process is effectively *measurable* with the dependability metrics mentioned before.

Techniques that enable failure-free execution or improve the probability of it and enhance the five attributes of dependability are categorizable into four *means* [Avi+04]:

Fault Prevention Fault prevention is a crucial aspect integrated into the system's design process. Techniques falling under this category aim to prevent the initiation of faults within the system, effectively preventing the initiation of the fault propagation chain and averting system failures. An illustrative example of fault prevention is manufacturing chips packaging using radiation-resistant materials [Con03].

Error Tolerance[6] Error tolerance becomes relevant during system execution. When a fault is unavoidable, an *error-tolerant* system can still perform its intended task without issues. To achieve this, we distinguish between error *detection* and error *correction*. Error-detecting techniques ensure errors are detected, and systems execute appropriate actions after detection. For instance, a simple error-detecting technique is using a *parity bit*. Error correction, on the other hand, directly repairs detected errors in real time, allowing the system to continue operating without failure. Failure-free execution can be, for instance, achieved through specific codes like *Gray code* and techniques such as *Error-Correcting Code (ECC)* [Muk08; Ros+11], *n*-modular redundancy with voting mechanisms [HHJ90; Yeh96; Sle+99; Rat04; Muk08], or resetting the system to a specific safe state.

Regarding the fault propagation chain, this approach ensures that if the chain has already begun, it will be interrupted along the way to the system boundary. Consequently, it prevents system failure or, at the very least, enables a reset to a safe system state, ensuring the system can carry out its tasks in the subsequent error-free execution.

Fault Forecasting This mean assesses the entire system, considering all fault prevention and error tolerance techniques, in terms of the types and quantities of potential faults, their origins, and the resulting consequences. This evaluation can be qualitative, involving the identification of fault origins and tracing fault propagation, or quantitative, which may include statistics on system failures such as FIT rates or probabilities of fault occurrences. One practical approach is injecting faults at a specific time and location within the system to analyze the resulting system behavior, known as *fault injection*.

Hardening[7] As a concept, hardening reduces the occurrence of faults or errors in the system during its runtime. Developers achieve this by preventing fault occurrences through thorough verification against the system's expected behavior, statically and dynamically. Additionally,

[6] The research community uses the terms *fault tolerance* and *error tolerance* often interchangeably, even though they represent distinct concepts within the fault propagation chain. Avižienis himself employs the term fault tolerance [Avi+04]. However, fault tolerance can also imply that faults may occur but have no effect or are not activated, a concept closely aligned with *fault prevention*. As tolerance approaches predominantly address activated faults, the terms error tolerance or error correction are more precise and appropriate.

[7] Similarly to the term error tolerance, the term "fault removal", as introduced by Avižienis [Avi+04], may not accurately capture the terminology related to the fault propagation chain. In practice, some fault removal techniques can eliminate errors, making distinguishing between fault removal and error removal challenging. Since both approaches contribute to a more dependable system by reducing the likelihood of fault occurrences or activations, I refer to this concept as *hardening*.

they diagnose the origins of potential faults and enhance fault-prone system components. The insights from fault forecasting are crucial in continuously improving fault prevention and error correction techniques.

Figure 2.6 shows the four presented means and their relations to each other. This dissertation's primary focus is *fault forecasting*, a critical aspect that allows us to assess and quantify the system's reliability. Fault forecasting plays a pivotal role in shaping hardening strategies, as the insights gained from this process directly inform decisions regarding fault removal and error tolerance.

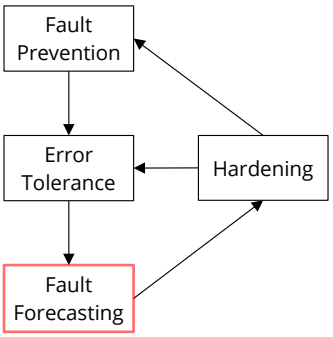


Figure 2.6 – Relations of the Dependability Means.
Fault prevention techniques prevent the activation of faults from becoming errors. *Error tolerance* techniques come into play once a fault is activated, preventing the propagation of the fault and thus averting a system failure. *Fault forecasting* facilitates the study of fault prevention and error tolerance in systems, providing qualitative or quantitative estimates of system reliability. The insights gained from fault forecasting play a significant role in the process of hardening, where fault prevention and error tolerance are improved.

The primary objective of *error detection* and *error correction* techniques is to ensure that the system never exhibits any form of misbehavior (refer to Section 2.1.2.1 on observed error trends and occurrences). In essence, error tolerance and correction techniques can effectively *break* the fault propagation chain (as outlined in Section 2.1.3) whenever an error occurs. This intervention is crucial for preventing the fault from propagating through the system's layers, ultimately averting a system failure. However, this work does not consider Fault Prevention further.

Error tolerance techniques can be implemented either in hardware or software. However, adjustments for error tolerance techniques in hardware become impossible when the system has a *fixed* hardware configuration. An alternative approach is to enhance the system's overall reliability by applying hardening on the software layer, mainly using techniques that fall under the SIHFT category.

This dissertation focuses on systems with fixed hardware components and versatile software components. It considers the effectiveness of integrated SIHFT techniques within the software layer to enhance fault tolerance capabilities via fault forecasting.

EXCURSUS: Error Tolerance and Error Correction Techniques

In this excursus, I will provide an extensive overview of frequently employed error-handling techniques, primarily derived from Schirmeier's dissertation's comprehensive compilation [Sch16]. These techniques can be categorized into two groups: hardware-implemented and software-implemented techniques. To fully comprehend these techniques, I will start with the concept of redundancy and delineate the various types of redundancy.

Redundancy

Redundancy serves as a foundational principle in error-handling techniques. It generally implies a higher resource use than the minimum necessary for a given function or objective, as defined by Koren [KK07]. In this context, Echtle [Ech90] identified four distinct approaches to implementing redundancy in systems:

Structural Redundancy This type of redundancy involves replicating or increasing the number of hardware components beyond what is strictly necessary for the core system function. For instance, an additional memory cell dedicated to a parity bit represents structural redundancy.

Functional Redundancy When a system includes non-essential supplementary functions that are non-essential for its regular operation, it is functional redundancy. An example is a parity check, which is not directly related to the primary system task.

Information Redundancy A system with information redundancy has extra data in various memory types that are not essential for the primary system task. The most common form of information redundancy is adding extra bits like the parity bit, a copy of data or coding, which adds check bits to the data, allowing the correctness of the data to be verified.

Temporal Redundancy This concept is related to additional runtime that extends beyond the time required for normal system execution. For instance, the system needs more time to check the parity bit, or it repeats the execution of its task over time, comparing each repetition results to identify any errors, which leads to additional CPU cycles as the core system's task needs.

Having redundancy alone does not inherently enhance a system's error tolerance, and the four types of redundancy are intricately connected. Take the example of a parity bit [Ham50]; initially, adding the bit does not offer any immediate advantages. Only when this bit is actively calculated and checked does the system gain error tolerance through the parity bit. However, this calculation or check comes with additional costs.
The bit needs more hardware or chip surface to store it, exemplifying *structural redundancy*. Additionally, extra functions are required to verify the parity, representing *functional redundancy* and extending the runtime (*temporal redundancy*). Finally, the parity bit is not essential for the system's core functionality, making its presence effectively redundant for the primary function, demonstrating *information redundancy*.
Ultimately, implementing error-tolerant mechanisms always involves a trade-off between cost and the desired error-tolerance level because achieving error tolerance is not without

expenses. Moreover, the example of the parity bit underscores the strong interplay among the four types of redundancy.
Although it may initially seem wasteful, the redundancy principle is quite common. Here are some examples of redundancy in nature and society:

- Numerous creatures on our planet possess multiple organs, providing a built-in redundancy. Take humans, for example; we have two lungs and two kidneys. This redundancy ensures that even if a person were to lose one lung or kidney due to an accident, the vital functions necessary for sustaining life would still be fulfilled, albeit in a diminished form.
- Climbing is a widely practiced hobby, and safety measures are paramount. Safety ropes are employed to prevent climbers from falling, depending on the type of climbing. These safety ropes typically contain with double hooks, which means that even if one hook were to loosen during a fall, the climber would still be securely anchored by the redundant second hook, ensuring their safety.
- In high-security facilities like prisons and banks, access points are often fortified with multiple doors. Although technically, only one door is necessary to secure a prison section or bank vault, redundant doors significantly increase the difficulty level for unauthorized individuals attempting to gain access. In practice, these access doors are never simultaneously open. Instead, they are designed to be opened one after the other, ensuring that the next door only opens when the previous one has been securely closed. This sequential arrangement adds an extra layer of protection to these critical areas.

Hardware-Implemented Error Tolerance

The initial layer of implementing error-tolerance techniques is their direct integration into the hardware. It requires extending or modifying the hardware to inherently lower the chances of an activated fault or propagated error. Implementing these techniques at the hardware level aims to create an illusion of error-free hardware execution.
In an ideal scenario, this approach would relieve software developers of concerns regarding transient hardware faults, as they could safely assume the correct execution of the hardware.

Physical and Digital-Logic Layer

Over the years, various techniques have been developed to minimize the probability of SEU occurrences. One widely adopted technique is the use of *triple-wells*, as introduced by Mukherjee [Muk08]. These triple-wells divert electrons during transient effects, which might otherwise erroneously switch a transistor.
Another method that aids in this context is the *silicon-on-insulator*, which helps reduce the accumulated charge during this process. Although both triple-wells and silicon-on-insulators contribute to increased production costs, they have become prevalent techniques for effectively decreasing the likelihood of transient faults. It, in turn, also enhances transistor performance [Muk08].
At the circuit layer, dependability is enhanced using memory cells containing redundant transistors, offering more reliable compensation for transient effects [Muk08]. This technique

significantly reduces the probability of transient faults, albeit at the cost of increased power consumption, requiring nearly double the chip surface. Given the associated cost increase, implementing such memory cells system-wide may not be feasible. Instead, the focus should be on critical areas within the hardware.

Memory Protection

In a broader system context, it is also possible to implement more cost-effective error-tolerance techniques at the level of interconnected 1-bit memory cells, such as SRAM and DRAM. An enduring and straightforward example is the previously mentioned *parity bit* [Ham50].
The parity bit stores a *zero* if the data contains an even number of *ones* and a *one* if it contains an odd number, a property known as parity. The parity is periodically compared with the parity bit, and if a mismatch is detected, it signifies a bit flip in the data, potentially compromising further data calculations [Muk08].
Error-Correcting Code (ECC) [Muk08; Ros+11] is another widely used technique to safeguard data, even in high-end computing. ECC detects bit flips and corrects them on the fly, allowing uninterrupted execution, which is the case with ECC-DRAM modules or ECC-protected CPU-cache SRAM. Typically, ECC implements protection according to the *Single-Error Correct Double-Error Detect (SECDED)* scheme, enabling the detection of double-bit flips or the correction of single bit flips immediately.
However, the SECDED scheme requires additional redundant bits per machine word, for instance, eight bits per 64-bit machine word. This results in an additional memory requirement of 12.5 percent, increased energy consumption, and longer memory access times [Muk08; Ros+11]. Some protection schemes can even correct more than a single bit, increasing overheads and intensifying the trade-off between performance and reliability [Del97; YE10; Ros+11].

n-Modular Redundancy

Another approach to safeguarding and validating data and the execution flow of a system is incorporating structural redundancy in the form of *n-Modular Redundancy (nMR)*. In a *Dual-Modular Redundancy (DMR)* two CPUs execute the same code synchronously. A *voter* component then compares the state of the CPUs or their results. An error is detected if a discrepancy in the results occurs between the CPUs [HHJ90; Sle+99]. This detected discrepancy should lead to reactions such as reboots, rollbacks, complete shutdowns, or similar actions.
The triple version, *Triple-Modular Redundancy (TMR)*, is also widely used in safety-critical domains, such as avionics [Yeh96] and spacecraft [Rat04], where the voter disregards the one faulty CPU and accepts the states or results of the two remaining CPUs. However, each nMR type leads to significant reductions in execution performance, increased chip surface, higher complexity in implementing the voter component, and greater energy consumption [Muk08]. This results in the well-known trade-off between performance and reliability.

Hardware Multithreading

Another approach to enhancing error tolerance is to replicate execution through techniques such as pipelining, chip-level multiprocessing, or simultaneous multithreading [TEL95].

Rotenberg [Rot99] implemented a CPU incorporating simultaneous multithreading and hardware-based check mechanisms. In this system, execution occurs a second time after a short delay, and the results are checked for equivalence, resulting in a speed overhead of 10-30 percent. Similarly, *lock-stepping* [Bal+03; AKL12] synchronizes multiple hardware threads to execute exact instructions simultaneously. The thread states are compared at predefined checkpoints, allowing them to react if the thread states are diverse.
Further developments of this principle have led to recovery techniques [VPC02]. Another approach, known as the *Simultaneous and Redundantly Threaded (SRT)* processor, duplicates executions only when memory accesses occur [RM00], which results in performance improvements [Rei+05b]. In this context, Reinhardt and Mukherjee introduced the principle of the *sphere of replication* [RM00], which suggests that components within the sphere of replication receive complete fault coverage through redundant executions, whereas components outside of it do not. Data that crosses this boundary requires replication and checking to maintain this protective sphere, as implemented in the SRT processors.

Software-Implemented Error Tolerance

In the hardware manufacturing process, hardware-implemented error-tolerance techniques have higher manufacturing costs, increased chip surface requirements, and greater energy consumption, and they often exhibit reduced performance compared to their native hardware counterparts. These reliability-enhancing drawbacks are more likely encountered in highly dependable server systems, where techniques like ECC or CPU lock-stepping [HHJ90; Sle+99] are used. Reliability-enhancing techniques like ECC or CPU Lockstepping [HHJ90; Sle+99] are typically used in highly dependable server systems but can also lead to certain drawbacks. In contrast, *Software-Implemented Hardware Fault Tolerance (SIHFT)* techniques offer greater flexibility, sustainability, and cost-effectiveness compared to their hardware-implemented counterparts, and they do not rely on specific hardware features. SIHFT leverages the knowledge of the specific application to be executed by the system. Whereas hardware-based techniques like ECC protect *every* bit, SIHFT allows for targeted and *flexible* protection of specific parts of the control flow or individual data structures. Additionally, fundamental principles like nMR can also be implemented in software, enabling *specific* segments of the program flow to be executed multiple times according to nMR rather than duplicating the entire execution through fixed physical replication in the hardware.
SIHFT techniques can be easily replaced, maintained, or further developed, which is impossible with pre-made hardware due to the shorter development cycles of software. If reliability requirements for the system change, SIHFT allows for system upgrades. Despite these advantages, SIHFT techniques are often slower and more energy-consuming than their hardware counterparts. Some hardware parts, like shadow registers, cache-line tags, or pipeline stages, cannot be protected at the software layer because they are not visible at the software layer and thus cannot be protected with SIHFT, making them vulnerable to errors. As with hardware-implemented error tolerance, a trade-off exists between performance, flexibility, cost, level of abstraction, and reliability when implementing SIHFT techniques.

Inherently Error-Tolerant Algorithms

Some algorithms are *inherently* error-tolerant, eliminating the need for explicit SIHFT. Numerical methods are examples of such inherent algorithms: Finding roots of continuous

mathematical functions (e.g., the bisection method and Newton's method), optimization algorithms for locating extremal points in nonlinear, multidimensional, continuous mathematical functions (e.g., pattern search and partial swarm optimization), and generic algorithms for discrete optimization.
Inherent error-tolerance mechanisms rely on the function's properties or the approximating algorithm's nature. For instance, if an SEU corrupts a function value found by one of these algorithms, it can often be detected based on properties of the function (e.g., exceeding function bounds or an incorrect sign) or bypassed due to the algorithm's inherent characteristics.
Artificial neural networks are another example of inherent error tolerance, as they can compensate bit flips through their adaptive properties. They have been used in the spacecraft domain for research satellite applications, which are notably vulnerable to bit flips [Vel+99].
However, the field of *algorithm-based error tolerance* has emerged, developing error-tolerant variants of various algorithms, including matrix multiplication [HA84; Tyr96], number factorization [Li+13], and sorting and searching algorithms [FI04; FGI05; FGI07; FI08; FPFI09; FGI09].

Hardware Detectors for SIHFT

When the inherent properties of code no longer suffice, system designers must implement explicit SIHFT techniques. One fundamental approach is using *detectors* that trigger under specific conditions. Hardware detectors, such as CPU *traps* or *exceptions*, are primarily designed to detect erroneous code execution, such as dividing by zero, executing malformed instructions, or invalid memory accesses.
However, these detectors can also be employed in reverse to identify potential errors in the hardware and trigger software-level responses. The research community has explored this concept, in several works [Mir+92; AN97; Alk+99; Rei+05b; Hof+15]. For instance, one approach is to populate unused memory with instructions explicitly designed to trigger traps. An error is promptly detected if the program accidentally jumps to this memory location due to a SEU-induced malfunction, [Mir+92; Hof+15].

Executable Assertions

Another method involving a "detector" is the use of *executable assertions*. These are statements that, at runtime, check *statically* known, *self-consistent* system states at *defined points* within the program [Sai77; MAM84; RBQ96]. Initially designed for identifying software bugs, assertions have proven effective in detecting data-flow errors and can also be used to identify hardware errors [Gol+06], thereby preventing error propagation [Sai77].
Executing executable assertions depends on a deep understanding of the program. Rabéjac and colleagues [RBQ96] proposed an informal process for placing assertions. On the other hand, Hiller and colleagues introduced a more systematic approach that categorizes data, including internal variables, and identifies misbehavior through generic assertions [Hil99; Hil00].
The effectiveness of executable assertions has been widely studied [CRA06; SK08a; Gaw+09; LJ11; HAN12], with detection improvements ranging from 0.23 percent [CRA06] to 100 percent [HAN12] in identifying corrupted data under various configuration setups. However, it is essential to note that using assertions comes with certain drawbacks, such as increasing

the static code length by 5.2 percent [SK08a] to 15 percent [Gaw+09] and extending runtime by up to 35 percent [LJ11].

Information Redundancy

Assertions verify data-flow errors and determine if a *consistent* system state has been achieved based on *known* system states. However, ensuring that this consistent state is *error-free* requires additional logic, achievable by incorporating *information redundancy*.
The simplest form of information redundancy is data *replication*. When protected data enters the system, it is duplicated, and if any data modifications occur, the duplicate updates as well. The data and its copy are checked for value equivalence during data reads. An example of automatic implementation is the source-to-source transformation tool *Reliabile Code Compiler (RECCO)* for C/C++[Ben+00; Ben+01b]. RECCO selects a subset of variables, duplicates them, and inserts checks to verify their equality. A customized error-propagation metric can be used to select variables for duplication [XTS08].
Hardware-based encodings like ECC (see Section 2.1.5) can also be implemented at the software layer. Shirvani and colleagues [SSM00] explored different variants of ECC and applied them on the ARGOS satellite [SM98; Shi+00; Lov+02]. *Arithmetic encoding* is one specific variant of encodings, which was initially implemented in hardware and preserved during arithmetic operations [Avi+71; Avi71]. This encoding is applied directly to the data without the need for decoding. In its basic form, the encoding algorithm multiplies a constant value A with each data word N system-wide. A data word is considered uncorrupted by an SEU if the remainder of the division of the data word N by A is zero. This encoding is also known as *AN encoding*, with successors like *ANB* and *ANBD encoding*. The advantage of this approach is that the encoding remains valid after various arithmetic operations due to the distributivity of addition and subtraction or even in another encoding variant, multiplication, division, and logical operations [Ulb14]. Another derivate of arithmetic encodings is the δ-encoding [KF14], which combines AN encoding and duplicated instructions.
Various works [WF07; FSS09; Sch+10a; Sch11] have explored potential implementations and values for A, as well as the weaknesses of arithmetic coding [Hof+14b; Hof+14a]. A notable weakness of arithmetic coding is the high overhead compared to simple data duplication. Although duplication may require up to 177 percent [Ben+00] more memory space and run up to 173 percent slower [PGZ08], arithmetic coding can slow down by a *factor* of up to 385 [Sch+10a] but does not require extra memory space. However, the fault coverage increases from 99.7 percent [XTS08] to 99.97 percent [Sch+10a]. The δ-encoding exhibits a fault coverage of 99.997 percent with a performance slowdown of up to a factor of 4.
Given the increased overhead associated with all information-redundant techniques, careful consideration is necessary to determine which of the *most critical data* should be protected [Sch16], as complete protection using these techniques may not be feasible.

Redundant Multithreading

Another SIHFT technique that can be adapted from hardware to software is code replication, similar to the concepts of SRT processors or nMR discussed previously. This approach, called *Redundant MultiThreading (RMT)*, functions at the software layer and focuses on temporal redundancy as equivalence checks and voting logic within the software code.

One well-known RMT implementation is *error detection by duplicated instructions (EDDI)* [OSM02]. EDDI duplicates and executes each instruction using copies of variables stored in different registers. Before writing data back into memory or executing conditional jumps, the duplicated executions are cross-verified for consistency. Although modern processors can execute duplications simultaneously, RMT introduces a runtime overhead of up to 113 percent [OSM02]. This overhead results from the functional complexity and access delays. However, the fault coverage achieved with RMT is nearly 100 percent [OSM02; Rei+05a; CRA06].
Coarser-grained solutions include duplicating function parameters as input for function calls in the C programming language [LH07]. Ulbrich, Hoffmann, and their colleagues have developed a task-level hardened voter for control applications [Ulb+12; Hof+14b; Hof+14a]. Some microkernels, including L4/Fiasco.OC, have integrated RMT at the application level [DHE12; DH12; DMH14; Doe14]. A variant of this approach can even handle multithreaded applications [Doe14; DH14].
RMT should be used cautiously, regardless of the level of detail applied. Despite its high parallelizability, RMT techniques introduce an overhead of more than 100 percent [Shy+07; Shy+09]. This overhead comes from inherent thread overhead, input data replication, and the voting process's cost. Additionally, RMT consumes more resources, which could be used for other purposes or lead to delays in subsequent executions. In some cases, RMT may not be feasible on single-core CPUs without a multistaged pipeline [Shy+07; Shy+09].

Control-Flow Checking

In addition to SEUs causing corrupted data and erroneous results, they can also lead to fetching a different instruction than expected [Gol+06]. This scenario results in *Control-Flow Errors (CFEs)*. Various factors can contribute to CFEs, including corrupted jump addresses and their calculations, bit flips in function pointers, and the direct influence of an SEU in the instruction pointer.
Control-Flow Monitoring (CFM) techniques monitor instruction sequences and identify CFEs. Most CFM techniques use the *Control-Flow Graph (CFG)* to represent a program's execution structure [All70; Gol+06]. The CFG contains nodes known as *Basic Blocks (BBs)*, each containing a fixed sequence of instructions. Edges in the CFG represent valid jumps between the last instruction of one BB and the first instruction of another.
A CFE occurs if a bit flip leads to the execution of a jump instruction that is either (1) not part of the set of all edges of the CFG, (2) the program jumps to an instruction in the middle of a BB, or (3) is a jump that is not expected at the current program state [Gol+06]. CFM monitors the program flow based on CFG edges and instruction sequence in a BB. It reports CFE on invalid jumps or unexpected instructions.
Miremadi and colleagues [Mir+92; MT95] describe techniques for computing a block-specific signature at the beginning of a BB and checking whether the signature remains consistent at the end of the BB. Ohlsson and Rim'en[OR95] use the CPU's watchdog timer and BB signatures to reset the CPU if a signature check does not occur within a specified time interval. Additionally, specially designed executable assertions exist to detect CFEs [Kan+96; AN97; Alk+99]. There is also an option for a source-to-source compiler developed by Benso and colleagues[Ben+01a], which automatically adds check code, models valid control flow using regular expressions, and reports a CFE if deviations are detected.

Depending on the specific monitored CFE and the desired fault coverage, CFM mechanisms can increase program code by 5 percent [RBQ96] up to 175 percent [YC80] and may extend runtime by up to 140 percent longer than without CFM [YC80].

Reliable Operating System

Existing software-based error tolerance techniques often focus solely on the application layer, overlooking the operating system, a crucial *Reliable Computing Base (RCB)* [ED12; Hof16] that forms the basis for all redundancy measures.
The C^3 microkernel [SWP13] monitors system-state transitions and recovers system components when a fault occurs. This approach assumes that faults will be immediately detected, which is not generally the case. On the other hand, L4/Romain [DH12] uses full system-call interceptions to achieve thread-level TMR, addressing the detection issue but still relying on the reliability of the microkernel in use. However, these approaches depend on the early and reliable detection of errors and their strict containment inside the RCB [Hof+15].
Conversely, the real-time operating system *Dependability-Oriented Static Embedded Kernel (dOSEK)* [HDL13; Hof+15] is a reliable operating system on potentially unreliable hardware. Unlike traditional approaches, dOSEK functions as an RCB and extends redundancy from the application to the kernel execution phase. Through three fundamental principles, dOSEK ensures reliable execution, preventing corrupted data: (1) By revealing a-priori application knowledge, dOSEK allows fine-grained system tailoring, contributing to a static system design. (2) These strategies aim to minimize the volatile part of the system state, minimizing indirections whenever possible and thus reducing the potential for errors. (3) dOSEK employs coarse-grained isolation and fine-grained arithmetic encoding for reliable error detection in the remaining vulnerable state.
Applying these principles leads to a remarkable reduction in the undetected corrupted data by four orders of magnitude compared to an off-the-shelf OSEK [Ose] operating system. Depending on the dOSEK variant, this reliability improvement comes at the cost of increased runtime of up to a factor of 8 and increased code size of up to a factor of 19 [Hof+15].

2.2 Fault Injection

As the adoption of SIHFT techniques continues to grow due to their cost-effectiveness, flexibility, and ease of maintenance, the need to assess and compare these techniques in a controlled and defined hardware environment is becoming more significant [Sch16]. *Fault Injection (FI)* is a widely used fault-forecasting method for analyzing the reliability of systems with integrated SIHFT and is a frequently researched topic in current studies.

Injecting a fault is possible in the actual physical hardware or a simulated hardware environment. Afterward, the system's resulting behavior is observed and categorized. The system's reliability and the effectiveness of the SIHFT techniques can be evaluated based on a predefined number of FI experiments.

I will start by introducing the *Fault Space (FS)* concept. The FS provides an overarching concept for understanding and representing potential faults in a system. It serves as a conceptual foundation for this dissertation.

Next, I will describe the fundamental process of a *FI campaign* and explain how individual FI experiments work conceptually.

After establishing this foundational knowledge, we will examine various existing FI techniques and their advantages and disadvantages.

Finally, we will narrow the focus of this dissertation and define a *fault model* that characterizes the specific type of fault under investigation.

2.2.1 Fault Space

I introduce the *Fault Space (FS)* concept that Goloubeva [Gol+06] proposed to provide a more precise description of fault occurrences within the system.

Figure 2.7 shows a universal representation of the FS schematically. The FS typically consists of two distinct dimensions:

Time The temporal axis begins at the point in time when we initiate the observation of the system, which could be the start of the hardware system's operation, the launch of a program, or the beginning of a program loop. It extends until the end of the operation or loop. The resolution of the time dimension can be defined according to specific requirements, such as CPU cycles, individual instructions, or discrete samples within the continuous time scale, often measured in seconds.

Location This dimension pertains to the possible parts of the hardware where a fault can occur. It can range from individual transistors to inputs of logical gates, bits within the register file, memory locations visible in the ISA layer, and more. The granularity can vary from single transistors to whole machine words or bytes, depending on the context.

This concept allows for a more detailed and structured analysis of fault occurrences within the system. It provides insights into both the spatial and temporal aspects of faults.

At each point in time along the temporal dimension, the system is in a specific *system state*. Here, a state represents the configuration of all the locations within the system at that particular moment. Each step along the temporal axis signifies a transition to the next state at the subsequent defined time point. At all possible time points, denoted as the set $T = \{t_0, t_1, \dots\}$, all possible locations define a system state, represented as the set $L = \{l_0, l_1, \dots\}$.

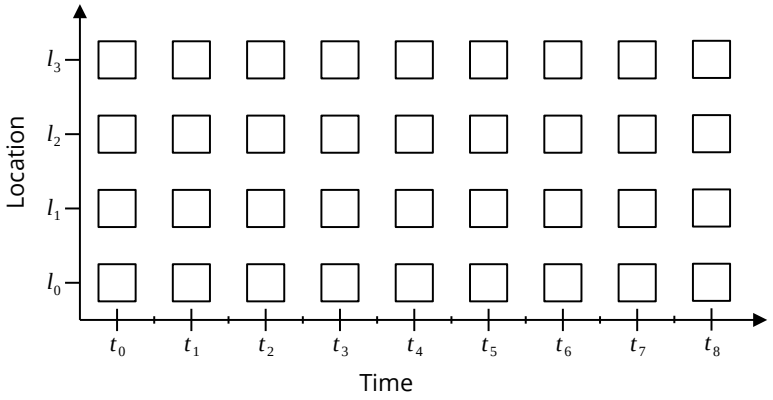


Figure 2.7 – Fault Space.
The fault space is a two-dimensional concept. On one axis, the time dimension can be considered at various resolutions, including seconds, CPU cycles, or individual instructions on the ISA layer. This axis can define the execution time of an entire program run or for a single program loop. On the other axis, it has the dimension of all locations within the system under consideration. Depending on the scope, these locations can represent individual bits, bytes, entire registers, and memory addresses. In essence, the fault space encompasses all possible combinations of the set of time steps T and the set of all locations L It describes all possible faults $\mathscr{F} = T \times L$ that could occur in the system. In this figure, $\mathscr{F}$ contains $|T| \cdot |L| = 9 \cdot 4 = 36$ points.

Let us consider the FS in the context of the ISA layer of a system, which includes components like register files and memory. All the *visible* bits of the registers (e.g., `r1`, `r2`) and the memory are withing the location dimension.

Time t_0 represents the system's *initial* state, as it has not yet started its operation. Assuming a resolution of single ISA instructions, such as `ADD r1, [r2]`, the system progresses one step forward along the temporal dimension. The execution of the `ADD` instruction is the transition from t_0 to t_1. These individual transitions are represented by the minor ticks along the temporal axis in Figure 2.7.

In this dissertation, I will equate the term *time point* (in any specific unit like instructions, CPU cycles, seconds) with the term *system state*; a time point corresponds to a state, and a state exists at one or multiple specific time points. With this understanding, the definition of the FS is the set of all possible faults that can occur in the system, denoted as $\mathscr{F}$:

$$\mathscr{F} = T \times L = \{(t_0, l_0), (t_0, l_1), \ldots, (t_1, l_0), \ldots, (t_{|T|-1}, l_{|L|-1})\}$$

Each point in the FS is a tuple of a time point and a location and stands strictly for one possible fault that can affect the system during its execution.

The FS depends on the specific system under evaluation, meaning that any alteration in the system, whether in hardware or software, results in an corresponding change to the FS. For instance, implementing an error tolerance technique (like the techniques presented in Section 2.1.5 or Section 2.1.5) to an existing system potentially expands the FS in both dimensions. Such implemented techniques typically lead to extended execution times and increased memory use. However, although these techniques generally enhance system reliability, they simultaneously enlarge the FS, thereby increasing the number of potentially occurring faults; thus, due to this fact, such techniques should be used cautiously.

2.2.2 Fault-Injection–Campaign Process

After defining the total number of possible injections, we can *systematically* explore the FS to conclude the system's reliability. This systematic exploration typically occurs within the context of FI campaigns, composed of individual FI experiments.

The objective of an FI campaign is to ensure that all points within the FS are covered, resulting in a complete assessment of the whole FS. In a naive approach, one FI experiment is conducted for every point in the FS. We will revisit this approach later in Section 2.3.2 to explore more efficient methods. Let us assume that one FI experiment per point in the FS exists.

In the following, I will use *System Under Test (SUT)* to refer to a system variant with SIHFT that has been modified for the evaluation process, whether in the hardware or the software.

Figure 2.8 illustrates a universal process for conducting a FI campaign. This illustrated process generalizes most of the procedures commonly used by existing FI frameworks. The following are the individual steps of this process:

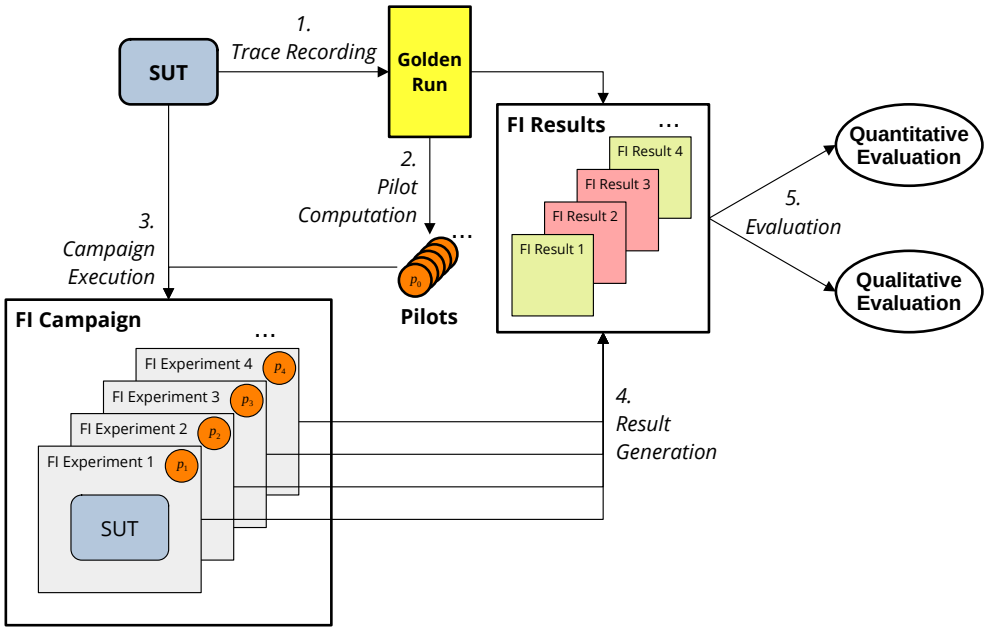


Figure 2.8 – Universal Fault-Injection–Campaign Process.
This illustration depicts a universal process for conducting FI campaigns. It begins with recording the program trace to generate a golden run, the reference execution of the SUT, and computing pilots. Each pilot corresponds to a single FI experiment, executed collectively in an FI campaign. Subsequently, the results from all FIs are generated and made available for evaluating the SUT.

Trace Recoding Recording the *program trace* is the initial step in the FI-campaign process. To establish a baseline for failure-free system execution, a reference, often also referred to as a *golden run*, is essential. During the golden run, the FI framework executes the SUT and records specific events of interest to generate a program trace representing the program's execution.

Events can encompass a range of activities, including changes in gate inputs, memory accesses, the execution of instructions, or alterations in the *Instruction Pointer (IP)*.

Listing 2.1 shows an exemplary program trace, as recorded by the tracing tool, which captures IP events and memory accesses at the ISA layer. Additionally, the trace includes register values, dereferences, and some metadata.

This reference trace is a foundation for evaluating subsequent FIs by clearly representing expected system behavior.

Listing 2.1 – Example of a Recorded Trace on the ISA Layer.
This snippet is from a recorded trace obtained while executing a qsort algorithm using a tracing tool for the ISA layer. The trace encompasses all IP and memory events that occurred throughout the execution of the qsort algorithm. To facilitate further analysis, additional details such as timestamps, access types, register values, and dereferences are also captured (note that "??" signifies that the memory is inaccessible with the provided register value).

```
IP 100052 t=76859622 REG 0 = 2101220 REG 3 = 2101500 REG 1 = 3223538018 REG 2 = 64733 REG 6 = 2101232 REG 7 = 0
↪  REG 4 = 2101052 REG 5 = 2101240 REG 17 = 2
MEM W 200f38 width 4 IP 100052 t=76859622 DATA 200ff8 REG 0 = 200fe4 -> fcdd REG 3 = 2010fc -> 0 REG 1 =
↪  c0234962 -> ?? REG 2 = fcdd -> 10000cd REG 6 = 200ff0 -> 10000 REG 7 = 0 -> f000ff53 REG 4 = 200f3c ->
↪  1003f0 REG 5 = 200ff8 -> 0 REG 11 = 2 -> ff53f000
IP 100053 t=76859623 REG 0 = 2101220 REG 3 = 2101500 REG 1 = 3223538018 REG 2 = 64733 REG 6 = 2101232 REG 7 = 0
↪  REG 4 = 2101048 REG 5 = 2101240 REG 17 = 2
IP 100055 t=76859624 REG 0 = 2101220 REG 3 = 2101500 REG 1 = 3223538018 REG 2 = 64733 REG 6 = 2101232 REG 7 = 0
↪  REG 4 = 2101048 REG 5 = 2101048 REG 17 = 2
MEM W 200f34 width 4 IP 100055 t=76859624 DATA 10005a REG 0 = 200fe4 -> fcdd REG 3 = 2010fc -> 0 REG 1 =
↪  c0234962 -> ?? REG 2 = fcdd -> 10000cd REG 6 = 200ff0 -> 10000 REG 7 = 0 -> f000ff53 REG 4 = 200f38 ->
↪  200ff8 REG 5 = 200f38 -> 200ff8 REG 11 = 2 -> ff53f000
IP 1004b0 t=76859625 REG 0 = 2101220 REG 3 = 2101500 REG 1 = 3223538018 REG 2 = 64733 REG 6 = 2101232 REG 7 = 0
↪  REG 4 = 2101044 REG 5 = 2101048 REG 17 = 2
MEM R 200f34 width 4 IP 1004b0 t=76859625 DATA 10005a REG 0 = 200fe4 -> fcdd REG 3 = 2010fc -> 0 REG 1 =
↪  c0234962 -> ?? REG 2 = fcdd -> 10000cd REG 6 = 200ff0 -> 10000 REG 7 = 0 -> f000ff53 REG 4 = 200f34 ->
↪  10005a REG 5 = 200f38 -> 200ff8 REG 11 = 2 -> ff53f000
```

Pilot Computation Computing *pilots* plays a crucial role in the FI-campaign process by aiming to identify all necessary pilots, essentially tuples of time and location in the FS of the SUT. These pilots represent individual FI experiments, and the goal is to ensure *complete coverage* of the FS. Covering the complete FS guarantees a corresponding statement regarding any faults in every situation in the SUT.

The golden run is essential in making the entire FS visible because the number of recorded events in the golden run determines the temporal axis's length in the FS. In contrast to a naive approach, which defines a pilot for each point in the FS, the golden run provides insights into which points require FI and which do not.

After the pilot computation, a set of pilots exists, ensuring complete coverage of the FS. This set of pilots is the foundation for executing FI experiments to evaluate the SUT's reliability.

Campaign Execution Individual pilots are employed to initiate single FI experiments in this step. The pilot specifies the time and location for executing the FI within the SUT. The SUT may incorporate additional elements like breakpoints, macros, or hardware evaluation boards to make FIs possible.[8]

The FI experiment commences by starting the SUT and continues until the execution reaches a predefined *time point* specified by the pilot. The experiment toggles the logical value at the

[8]The system is generally adjusted to enable tracing, often contingent on the tool used. For instance, it may employ redundant, application-independent function symbols within a C program to initiate and conclude tracing, thereby delineating the tracing's scope. Typically, the adaptations made for tracing or the SUT do not alter the system's semantics compared to its original behavior.

specified *location* when the SUT reaches this point (e.g., through a breakpoint). After this toggle, the FI framework monitors the SUT's behavior. The resulting behavior of the SUT *post-FI* constitutes the outcome of the FI experiment.

The collective sum of all experiments forms the complete FI campaign. Each experiment is performed independently, without data dependencies, allowing for a highly parallelizable process.

Result Generation In the result generation step, each experiment *reports* the SUT's *observed* behavior after the FI. The FI framework evaluates the observed behavior as equivalent to the golden run or divergent. More detailed *classifications* of deviations from the golden run are also feasible, such as identifying erroneous outputs, timeouts, or traps. FI frameworks typically store the results in a suitable data structure like a database. This campaign process is designable as a client-server architecture, where the individual FI experiments act as clients, and a server is responsible for managing the campaign's workload and storing the results.

Evaluation The final step in the process is the evaluation. For *each* pilot, there is a corresponding result that describes how the SUT behaved during the corresponding FI experiment. These results collectively provide a comprehensive understanding of the SUT behavior for *every* individual point in the FS. This complete assessment is a fundamental prerequisite for further analyses [BCS69].

One commonly used metric to gauge the reliability of a system is the fault-coverage factor [BCS69; Lev+09]. This metric is calculated by contrasting the number of failed experiments, denoted as f (i.e., those that deviate from the system behavior recorded in the golden run), with the cardinality of the FS, denoted as $|\mathscr{F}|$. The fault-coverage factor is defined by the formula $c = 1 - \frac{f}{|\mathscr{F}|}$. Consequently, fault coverage quantifies the probability of an SUT's ability to recover from a fault [BP03].

Additionally, related to fault coverage, the *Architectural Vulnerability Factor (AVF)* [Muk+03a; Muk+03b] measures the vulnerability of a specific hardware component of a system regarding a fault to be activated, which potentially leads to an error [Muk+03a]. Similarly, the *Program Vulnerability Factor (PVF)* [SK08b; SK09] defines the vulnerability of *"any architecturally-visible resource"* on the ISA layer, encompassing components like the CPU, registers, and memory.

Another approach is using the trace and the resulting system behavior to qualitatively analyze where the fault is activated and trace it back in the executed code. Additional metrics can be calculated to enhance this analysis, such as *fault latency*, which represents the time from when the fault occurs to its activation, or detection latency, which signifies the activated error to the observable system failure [BP03].

Indeed, the metrics mentioned previously in Section 2.1.4 are also applicable in this evaluation step.

The results of the FI-campaign process are eventually used to implement new SIHFT techniques or hardware mechanisms at suitable locations, thereby strengthening the system's reliability.

The FI-campaign process ultimately leads to implementing new SIHFT techniques or hardware mechanisms in appropriate locations to enhance the system's reliability. The resulting hardened system can undergo the described process repeatedly, potentially creating an iterative cycle that progressively enhances the system's reliability with each iteration.

As previously mentioned in Section 2.2.1, any modification to the system results in a potential FS alteration of the SUT. Consequently, the new FS must be used throughout the new iteration of

the FI-campaign process, which needs repeating tracing and pilot computation after each alteration to the SUT.

2.2.3 Types of Fault-Injection Techniques

The most straightforward approach is to run and test the system in its actual environment, using the real hardware configuration and software. However, field experiments in the natural environment are impractical due to the improbable and challenging nature of reproducing transient hardware faults [Hof16]. Consequently, various FI techniques have been developed to simulate the behavior of transient hardware faults. These techniques enable the evaluation of software execution on a hardware system regarding reliability, employing an FI-campaign process described in Section 2.2.2.

To thoroughly evaluate these techniques, I will introduce specific criteria and then categorize and assess individual FI techniques based on these criteria. The FI techniques presented in this section are primarily based on Schirmeier's dissertation [Sch16]; refer to this dissertation for more details.

I will present categories in which FI techniques can be classified [BP03], discuss the FI techniques within each, and evaluate them based on the established criteria. If I do not mention a criterion in the subsequent sections, the corresponding FI technique neither excels nor performs poorly concerning that criterion.

2.2.3.1 Criteria of Fault-Injection Techniques

Skarin defined criteria for assessing FI techniques [SBK10]:

Repeatability A fundamental requirement for obtaining accurate results in an FI experiment and, consequently, in the entire FI campaign is repeatability. This criterion implies that the results should consistently be the same when the experiment is repeated. Achieving repeatability requires a *deterministic* evaluation environment to ensure that the experiment's outcomes are consistently *reproducible*.

Controllability Controllability is the ability to execute a single FI experiment with precision. It involves manipulating a specific location in the SUT at a defined time, making it *mutable*. Additionally, executing a program on the SUT should be easily *interruptible* and *resumable* for full controllability, allowing precise control over the experiment.

Observability This criterion is crucial for obtaining accurate insights into the impact of individual FIs on the SUT. The more *observable*, *traceable*, and *measurable* the effects of individual FI events are, the more precise the conclusions regarding their influence on the SUT. This criterion emphasizes the importance of having detailed and comprehensive information about the behavior of the SUT during and after FIs.

Intrusiveness Due to requirements imposed by the FI technique, it is *intrusive* if the SUT deviates from the original system, either semantically or in terms of the codebase. Intrusiveness refers to the extent to which an FI technique must alter the properties or semantics of the system to enable FI. Intrusiveness can occur for technical reasons, even if they are unintentional, and are necessary to make the FI process possible. Examples of intrusive actions are modifying the program code or adjusting the memory.

Reachability That criterion refers to the *accessibility* of the individual FI locations planned in the FI campaign within the SUT at specific times. Access to these locations might not be possible due to factors such as restricted permissions in the SUT, abstractions between layers (e.g., shadow registers not being visible in the ISA layer), or technical limitations, such as missing pins in hardware.

In addition to Skarin's criteria, Schirmeier [Sch16] also considers the following criteria:

Cost The costs include both the *end-to-end runtime* of the evaluation and the *initial setup costs* of the evaluation environment. Additionally, it encompasses the ongoing expenses associated with maintenance and potential expansion or upgrades of the evaluation environment.

Scalability Scalability is close to costs. It refers to the evaluation process's ability to handle an increasing workload while maintaining *feasibility*. Scalability assesses whether the duration of the FI campaign remains *manageable* as the workload grows.

Realism Realism expresses how accurately the SUT and its environment represent the system under evaluation. It also considers how an FI technique models a fault and whether it realistically simulates it, aiming to reflect the actual system and its behavior *accurately*.

2.2.3.2 Hardware-Based Fault Injection

Evaluating a system directly in the field provides the closest representation of reality. However, in specific environments, faults are highly improbable, making practical evaluation challenging. To increase the probability of faults, techniques such as direct particle radiation [GKT89; San+14], explicit electromagnetic interference [Kar+94], or voltage fluctuations [Kar+91] can be employed. These methods align with reality, especially when the SUT remains unchanged. However, due to the *non-deterministic* nature of environmental influences, results from such experiments are neither deterministically repeatable nor easily controllable. Additionally, these experiments are time-consuming for achieving statistically significant reliability statements. Moreover, experiments with techniques like ray guns can be prohibitively expensive and potentially hazardous in the case of radioactive target environments [Kar+94].

Another method of injecting faults into hardware is setting bit patterns on the processor pins of the system [Mad+94]. Although the results of this technique are reproducible, ensuring repeatability, it suffers from poor controllability over the exact injection time. Additionally, changing the internal state of the SUT is possible only indirectly, and the alterations caused by the FI are challenging to observe. Observability and controllability have improved with debugging interfaces [Fid+06], but these experiments still face scalability issues.

Hardware-based FI techniques require fewer modifications in the SUT to represent the real system or apply the fault. However, these methods are often time-consuming and expensive, posing scalability, repeatability, and controllability challenges.

2.2.3.3 Software-Implemented Fault Injection

Software-Implemented Fault Injection (SWIFI) techniques offer a more cost-effective and scalable alternative in the context of FI. In SWIFI, program code, including the code or data segment, undergoes deliberate modifications to mimic faults. These code modifications occur either *before* the compilation or execution of the program or *during* the program execution itself. SWIFI exist in two variants *pre-runtime* SWIFI and *runtime* SWIFI.

Pre-runtime SWIFI injects faults before the SUT begins its execution. Initial pre-runtime SWIFI approaches used tools injecting bit flips directly into machine instructions [KKA95] or data and code segments [Fuc96; Aid+01; SBK10]. Modern techniques leverage the intermediate code of LLVM Intermediate Representation [LA04] for injecting bit flips into the program [Sch+10b; Sch11; TP13; Lu+15], ensuring independence from the underlying system.

Pre-runtime SWIFI offers cost-effectiveness compared to hardware-based techniques, enhancing scalability, repeatability, and controllability. However, it fails to mimic realistic transient fault effects, presenting activated faults primarily as bit flips. As injection occurs before execution, the entire temporal dimension of an FS condenses to a single pre-runtime time point. Consequently, the implementable FS becomes one-dimensional, emphasizing *only* the location. Hence, pre-runtime SWIFI techniques exhibit limitations in achieving high realism [Sch16].

SWIFI techniques operating *at runtime* exhibit greater realism and flexibility [Bar+90; KKA95; Car+95; SBK10]. For instance, controlled pauses in the SUT enable precise FI regarding location and time during runtime. Executing generic FI code empowers the technique to inject faults into the executing code, software interfaces [Koo+97; MBC10; Win+13; Win+15], or the kernel [KIT93; KI94] of the SUT with SEUs at runtime.

Runtime SWIFI achieves a higher level of reachability and a more realistic representation of behavior than pre-runtime SWIFI. However, it leads to a greater degree of intrusiveness, enlarging the considered FS unnecessarily due to additional workload. Despite this, runtime SWIFI remains a popular FI technique due to its lower cost than other FI techniques [Sch16].

2.2.3.4 Simulation-based Fault Injection

In contrast to SWIFI techniques, *simulation-based* FI techniques execute the program in a simulated hardware environment,[9] preserving the program's integrity to avoid increased intrusiveness [Koo+14]. The controlled simulation of hardware provides optimal controllability and observability over the SUT [STB97; Arl+02].

The heightened controllability and observability enable the extraction and, if necessary, restoration of system states. This capability helps conserve execution time for individual FI experiments and, consequently, for the entire FI campaign. This deterministic controlled simulation ensures that evaluations are consistently reproducible. However, the realism and accuracy of the simulation depend on the underlying hardware model, and simulations generally run slower than actual hardware.

The execution speed of simulations decreases as the hardware model becomes more low-level, involving more components and making it challenging to scale. Typically, SUTs are simulated at the ISA layer balance between accuracy and execution speed [Sch16]. The unavailability of lower-level hardware models may also dictate this choice. The technique is highly parallelizable due to the isolated execution of individual experiments in the simulation, providing a cost-effective alternative to running the entire FI campaign on costly prototypes.

2.2.3.5 Summary of the Fault-Injection Techniques

Field experiments are desirable for evaluating an SUT as they closely replicate real-world conditions. However, collecting results from these experiments can be laborious, and their reproducibility is challenging. The three categories of FI techniques aim to *emulate* the real-world behavior of SEUs, each with its own set of advantages and disadvantages.

[9] Alternatively, a virtual machine can execute the program. FI techniques based on virtual machines are also termed simulation-based FI techniques in the context of this dissertation, as the distinction between the two is marginal.

Among these categories, simulation-based FI is the preferred choice across various criteria (refer to Section 2.2.3.1). Table 2.1 summarizes the strengths and weaknesses of different FI-technique categories based on Schirmeier's appraisal [Sch16].[10]

I use a simulation-based FI framework In my dissertation, to capitalize on its numerous advantages.

	Hardware-Based	Software-Implemented	Simulation-Based
Repeatability	∘	+	+
Controllability	∘	+	+
Observability	∘	+	+
Intrusiveness	+	∘	+
Reachability	∘	-	+
Cost	-	+	+
Scalability	-	+	+
Realism	∘	-	∘

Table 2.1 – Overview of the Fault-Injection Techniques and its Assessed Criteria.
The table presents one FI technique category per column and one criterion per row. A plus sign (+) indicates that a technique category excels in the specified criterion. In contrast, a minus sign (-) suggests otherwise, and the circle (∘) represents neither a real advantage nor a disadvantage. The degree of realism for simulation-based techniques depends on the implementation of the hardware model. Despite this, simulation-based technologies generally outperform other categories in various aspects.

2.2.4 Fault Model

After outlining a basic understanding of FI techniques overall, the next step is defining a *Fault Model (FM)* that accurately represents the influence and nature of the faults on the SUT using a specific FI technique. The primary objective is to establish a complete set of potential faults that could manifest within the SUT. Firstly, I describe the components of the FM used in this dissertation and conclude with a concrete FM summary.

2.2.4.1 Components of the Fault Model

An FM specifies *how* a fault could impact the entire system, *when* a fault might occur, *where* the fault could be activated, and *how many* faults could affect the overall system.

The uncontrolled acceleration of the Toyota Lexus ES 350 in Section 2.1.3 exemplifies *how* a fault could affect a system. A fault in the physical layer can propagate and lead to unexpected behavior observed at the application layer. Error-tolerance mechanisms on the hardware or SIHFT can prevent this, as explained in Section 2.1.5.

The definition of *where* and *when* a fault occurs is related to each other and dependable on the *hardware model complexity* of the targeted system layer. Regrettably, due to its intricacy, it is *impractical* to create a simulation that encompasses an entire hardware system capable of tracking a fault from the physical layer up to the failure of the application layer. It is common to focus

[10] As Schirmeier's dissertation [Sch16] delves into the development of his own FI framework, it provides a more extensive description of FI techniques. For a more in-depth exploration, I recommend his work as a valuable source of additional insights in the domain of existing FI techniques.

exclusively on *one* layer to manage complexity and reduce simulation time [Cho+13]. Therefore, the initial step is selecting a specific layer for the FM.[11]

Employing low-level layer hardware models and corresponding simulations proves excessively intricate for FI techniques [Cho+13]. This approach is deemed *impractical* and *unsuitable* as a base for an FM. An additional drawback of using a low-level layer for the FM is the absence of comprehensive descriptions of available hardware technologies on the market. Conversely, considering higher layers leads to the abstraction of more parts or information about the system. Consequently, simulations become progressively less accurate the higher the selected system layer is. For instance, accessing individual memory cells at the microarchitecture layer is not possible, and on the ISA layer, shadow registers remain inaccessible. Figure 2.9 illustrates this trade-off.

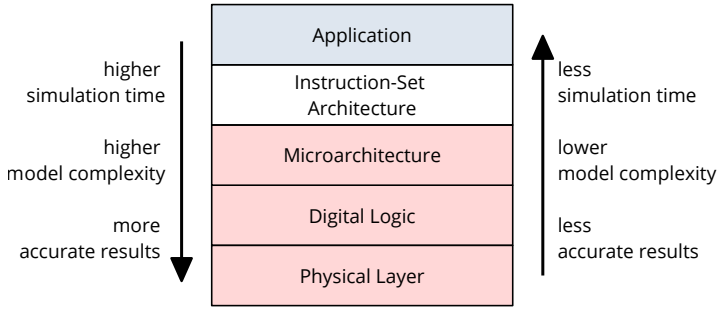


Figure 2.9 – System-Layers Evaluation Trade-Off.
Faults inherently occur on the physical layer. Although analyzing all conceivable faults at this layer would represent reality, it proves impractical due to the substantial time investment involved. Opting for higher levels of abstraction maintains a lower hardware model complexity and simulation time. Nonetheless, each subsequent abstraction diminishes information about a system layer, leading to higher inaccuracy. The choice of the considered layer leads to the trade-off between the required simulation time, hardware model complexity, and the accuracy of the simulated hardware.

High-level system models, such as the ISA layer, are commonly employed in simulation-based FI [Sch16]. The ISA layer serves as an interface bridging the low-level hardware layers and the system's executed software. Errors in structures visible on the ISA layer emulate the propagation of faults in the abstracted lower-level layers. For instance, a fault on the physical layer propagates to a bit flip visible in the ISA layer in registers or memory (representing the location dimension of the FS). The execution of an instruction triggers each step in the SEU's propagation (representing the temporal dimension of the FS). The ISA layer provides a fixed hardware configuration, including hardware-based error-tolerance mechanisms. Consequently, the ISA layer proves suitable for evaluating software and its SIHFT on fixed hardware, facilitating the analysis of the overall system reliability in a simulation-based FI-campaign process. Additionally, executed software only reads out or alters system states in the ISA layer, not typically in any lower system layers. Furthermore, several ISA-layer models and simulators are well-known in the research community, often subject to ongoing development and maintenance, and frequently available as open-source tools. Examples for such tools are the open-standard ISA RISC-V [WA19; WAH21], the x86 ISA simulator Bochs [Law96; Pro23a], and the gem5 computer architecture research project [Bin+11; LP+20; Pro22].

[11] Additionally, it is possible to formulate cross-layer fault models to examine the impact and propagation of SEUs across multiple layers. Whereas cross-layer approaches represent a research focus within the academic community [XL12; Ent+12; Gla+15; Val+15; Bar+17; Fle+17; Che+18], these are not within the scope of the current discussion.

The caused behavior specified for transient faults is non-deterministic, capable of occurring at any time and anywhere in the system. The research community has validated this assumption, observing a uniformly distributed probability of occurrence for SEUs in memory, such as SRAM [DW11; Hen+13] and DRAM [Sri+13]. This phenomenon is quite common in systems [Hen+13; Mez+15]. Therefore, it is a widely accepted assumption for FMs in the FI research community, particularly for ISA layer FI.

In contrast to memory faults, modeling CPU faults is more complex [CS90; Kan+96; Kal+98; Kar+08; Man+11; Kor12; Cho+13]. Faults in sequential circuits of the CPU manifest as SEUs in registers but also lead to more complex effects, such as multi-bit flips [Cho+13], interrupts [Kor12], or deadlocks [Kor12]. As a simple approximation, FI frameworks often emulate CPU faults with bit flips in CPU registers [HSR95; TI95; Ben+98a; Cin+09; SBK10; DeB+12; LT13; TP13; DC+14; Sch+15; Sch16], assuming a uniform distribution of occurring faults [HSR95; PRSR98], as it is in this dissertation.

In the context of the general FI-campaign process (refer to Section 2.2.2), I assume that there is precisely one FI per FI experiment based on the set of points in the considered FS (refer to Section 2.2.1). Lastly, I address the single-fault assumption of this FM, which implies that exactly one fault will occur at a time. This assumption remains valid since even a single flip of a specific bit between two instructions is highly improbable in reality [Li+10; SL12; Sri+13].

The probability of a fault occurring in a single system execution is well approximable by a Poisson distribution [Tri08], assuming the occurrence of faults follows a Poisson process [MW79; Nor96b; Li+07]. According to the Poisson distribution, the probability of a fault occurring in a single system execution is only $1.328 \cdot 10^{-11}$ percent. The probability of two faults co-occurring is $8.821 \cdot 10^{-25}$ percent, and three or more faults co-occurring is much lower. Thus, whereas the dominant case is that no fault occurs, with a probability of more than 99.9999 percent, the probability of two or more faults occurring is negligible due to the extremely low probability. Therefore, the single-fault assumption is sufficient for the FM.

However, single faults occurring at the physical layer can result in a bit flip through propagation to the ISA layer, as previously highlighted. Due to the structure of the system hardware and its layers, a single fault at the physical layer can induce a *Multiple-Bit Upset (MBU)* observable on the ISA layer through propagation (i.e., more than one flipped bits become visible on the ISA layer due one single fault).

Systematically traversing through the ISA layer FS (refer to Section 2.2.1) covers *all* potential bits of *any* MBU pattern within the single FIs campaign. Consequently, the influence of the individually flipped bits within each bit of the MBU on the SUT is known. According to the research of Li and colleagues [Li+10], flipped bits of MBUs tend to occur adjacently, often in a word-wise manner. Therefore, single bit flips can potentially cover resultant failure classes that might occur with multiple adjacent flipped bits. For instance, flipping bits in a pointer can lead to erroneous, invalid memory accesses, potentially irrespective of their number.

In the context of this dissertation, the examination is focused solely on individual bit flips, which serve as representatives of bit flips and MBUs. The limitation of this assumption is that when individual bit flips in FI experiments are benign, multiple bit flips simultaneously lead to non-benign behavior. However, the criticality of this scenario will be elaborated upon in detail in a later part of this dissertation (Chapter 7).

2.2.4.2 Fault-Model Summary

In conclusion, I provide the following list of all the specific components of the FM considered in this dissertation:

- Faults manifest at the physical layer and *propagate* through various layers, eventually leading to *externally observable behavior* at the application layer (refer to the fault propagation chain in Section 2.1.3) when not benign.
- The system layer under consideration is the *ISA layer*, acting as the interface between hardware with fixed configurations or manufacturing and software that includes implemented and adaptable SIHFT.
  - Executing the next instruction may propagate an error between two instructions. Each instruction alters the system's state, advancing the system temporally by one step.
  - Possible *locations* where a bit flip can occur are a system's *main memory* and *CPU registers*. These locations are observable and changeable in the ISA layer, which is crucial for software development with SIHFT.
  - The probability of fault occurrences is uniformly distributed over time (instructions) and location (memory and register bits).
  - An error at the ISA layer *emulates* the propagation of a fault from the physical layer to the ISA layer, manifesting as a single bit flip in the system.
- The probability of *two or more faults* co-occurring and *arbitrary* MBU patterns occurring is extremely low.
  - To evaluate the reliability of a system exposed to transient hardware faults, it is reasonable to assume that only *single* faults occur.
  - An MBU's bit flips are mainly neighboring [Li+10]. Systematically traversing the FS within an FI campaign, explicitly targeting the ISA layer FS, comprehensively covers the impact of individual bits within MBUs. Consequently, the FI campaign exclusively focuses on considering *single bit flips* per FI experiment, which emulates the influence of single SEUs on the physical layer.

The choice of FI technique determines the specific FM and implicitly establishes the FS. In the context of this dissertation, the FS outlined in Section 2.2.1 is designed for simulation-based FI, guided by the FM presented here. Planning and executing an FI campaign becomes possible with a well-defined FM and corresponding FS for the system under evaluation and the employed FI technique.

Conclusively, clarifying the term *fault* injection in that context is essential. In injecting on the ISA layer, a "fault" technically is an *error* visible on the ISA layer within the fault propagation chain (refer to Section 2.1.3 on page 19). However, "Fault Injection" is commonly used universally across all layers in the research community, . For consistency within this dissertation, "fault injection" consistently refers to a bit flip specifically occurring on the ISA layer.

2.3 Accelerating Fault-Injection Campaigns

The primary objective of this dissertation is to enhance the overall runtime of FI campaigns, (approximately) denoted as $t_{\text{cpn}} = n \cdot t_{\text{exp}}$, where n is the number of pilots and t_{exp} is the FI-experiment runtime. Whereas the main focus is on the acceleration of FI campaigns, it is worth noting that the improvement and evaluation of the SUT reliability is not directly the focus of this work. However, a more rapid FI campaign has the potential to run through additional iterations of the assessment cycle within the same time frame compared to an FI campaign without any acceleration methods.

When opting for a specific FI technique, the implementation of a corresponding FS corresponding to the FM allows for the execution of an FI campaign. However, issues arise when processing the FS naively, as signified in the FI-campaign process in Section 2.2.2, without incorporating additional logic. The naive approach treats *each* point in the FS as a separate FI experiment. In the case of most benchmarks and realistic programs, the native approach becomes impractical, leading to what I term an *FI-campaign runtime explosion*. This section addresses the challenge and discusses methods available to mitigate the total runtime of FI campaigns. These methods can be applied either *before* initiating the FI campaign or *during* the execution of individual FI experiments to counteract the runtime explosion effectively.

2.3.1 Fault-Injection–Campaign Runtime Explosion

I discussed the purpose of FMs (refer to Section 2.2.4), the corresponding FSs (see Section 2.2.1), and various FI techniques that ultimately generate pilots from an FS to create FI experiments. Once the appropriate FI technique is selected, the FI campaign can be prepared and executed.

Every single point in the FS could be a pilot. However, adopting such a naive approach is practically unfeasible. For instance, assume a scenario with an ARM Cortex-M33FU5 processor [Armb] executing one loop of an ABS routine in a car that takes $t_{\text{prg}} = 10\,\text{ms}$. The resulting FI-campaign runtime t_{cpn} when injection in every single cycle would be at least 1.2 years. Even if this calculation is extrapolated to instructions,[12] it is *not* practically feasible to examine every point in the FS using FI experiments (more than a month FI-campaign runtime).

The exemplary necessary FI-campaign runtime of approximately 1.2 years includes no temporal overhead t_{ov}. It represents the pure summed-up runtime of all possible program executions in the FI campaign, serving as a lower bound. The actual runtime of the average FI-experiment, denoted as $t_{\text{exp}} = t_{\text{prg}} + t_{\text{ov}}$, is even higher. Consequently, the overall FI-campaign runtime will exceed 1.2 years.

Even with small processors and short program runtimes, the FS can quickly become impractically large, making it unfeasible to evaluate in a naive way. Injecting at each cycle or the instruction level (as defined in the FM of this dissertation) is a minor factor. It does not alleviate the problem of the *fault-injection–campaign runtime explosion* in practice.

When calculating the total FI-campaign runtime $t_{\text{cpn}} = n \cdot t_{\text{exp}}$ with n pilots, two factors influence the campaign's runtime [Gol+06] (without considering additional overheads): (1) the number of pilots required to cover the FS completely, denoted as n, and (2) the average runtime for a single FI experiment in the campaign, denoted as t_{exp}.

The extensive size of the FS makes it impractical and resource-intensive to execute FI experiments for every individual point. Consequently, strategies to minimize the number of experiments or optimize the FI-experiment runtime become imperative. The following sections explore methods to

[12] The ARM architecture specifications do not specify for every processor how many cycles each instruction should take to execute because it is implementation defined [Armb]. Refering to a manual from ARM [Arma], instructions with a lower complexity need 1 to 2 cycles per instruction. Assuming a high average number of 10 cycles per instruction, the potential lower bound for the campaign duration with instructions on the FS's temporal dimension would still be more than a month.

reduce these two factors, n and t_{exp}, aiming to lower t_{cpn} and make FI campaigns more feasible and efficient.

EXAMPLE: Lower-Bound Calculation of an FI-Campaign Runtime

The ARM Cortex-M33FU5 operates with a maximum clock frequency of $f = 160\,\text{MHz}$ and possesses a total memory size of $s_{mem} = 2048 + 768\,\text{KiByte} = 2816\,\text{KiByte} = 23\,068\,672\,\text{bit}$ (combining flash and SRAM).
The ABS in the car must efficiently process all signals required for decision-making within a response time that does not exceed a given threshold. Let us assume a 10 ms threshold, establishes an upper bound for the program, specifically its loop, at $t_{prg} = 10\,\text{ms}$.
With a cycle time of $t_{cycle} = \frac{1}{f} = 62.5\,\mu\text{s}$, the ARM requires $c = \frac{t_{prg}}{t_{cycle}} = 160$ cycles to execute one program loop entirely. Considering the number of cycles needed for one execution (c) and the number of bits (s_{mem}), the complete two-dimensional FS size is $|\mathscr{F}| = c \cdot s_{mem} = 160 \cdot 23068672 \approx 3.691 \cdot 10^9$ possible injections.
However, without any further logic, the number of pilots n equals the FS size $|\mathscr{F}|$ to ensure complete coverage of the FS.
Every program execution must conclude to observe the system's behavior after an FI. Therefore, with $|\mathscr{F}| = n$ injections and the average runtime of a program loop t_{prg}, the cumulative program execution time for the FI campaign is (at least)

$$t_{prg} \cdot n = 36\,909\,875.2\,\text{s} \approx 1.2 \text{ years}$$

In this calculation example, the temporal resolution is the level of individual cycles.

2.3.2 Pre-Injection Analysis

In this section, we explore strategies to reduce the number of FI experiments, aiming to decrease the overall runtime of the FI campaign. As outlined in Section 2.2.1, it is essential to analyze the effect of each potential FI within the entire FS. To completely address the FS while minimizing the number of pilots required and overall FI-campaign runtime, acceleration methods exist that can precisely or approximately handle the FS.

In evaluating the methods employed to minimize the number of pilots, this work emphasizes two fundamental properties:

Fault-Space Completeness A method is *FS-complete* if it allows for a statement about the impact of bit flips for *every* point in the FS, indicating *full coverage* of the FS.

Precision of the Results The precision characterizes a method as precise if the resulting observable behavior of the SUT is *deterministically* reproducible or if there is no *approximation* of results. When heuristic methods are employed, the results of one FI experiment are approximated to another FI experiment based on the specific method, leading to a reduction in precision.

The central idea is grouping FIs into *Fault-Equivalence Sets (FESs)* that result in the same externally observable behavior of the SUT after the FI.[13] From these FESs, a single representative FI is selected

[13] As outlined in Section 2.2.4, the term "Fault-Equivalence Set" technically denotes an *error* equivalence (regarding the fault propagation chain, refer to Section 2.1.3), given that bit flips on the ISA layer represent manifested errors, which serve as an emulation of faults propagating from lower system layer up to the ISA layer. To maintain terminology consistency, when I refer to "Fault-Equivalence Sets", akin to the term "Fault Injection", I specifically denote a set of bit flips that share an equivalent influence on the SUT resulting behavior.

and executed. Once the outcome of this FI is determined, the result is extrapolated to all other potential FIs within the corresponding FES. Consequently, in a FI campaign, only as many FIs need to be executed as there are FESs; in other words, the number of pilots n equals the number of FESs.

2.3.2.1 Def/Use Pruning

Def/Use Pruning (DUP) is one of the most widely adopted methods. It stands out as a *precise* approach that ensures *complete coverage* of the FS. This well-known and foundational method was initially introduced by Smith and colleagues [Smi+95] and later revisited by Güthoff and colleagues [GS95]. Over time, it has been adapted to various system layers and FMs, with contributions from researchers such as Benso and colleagues [Ben+98b; BP03], Barbosa [Bar+05], and Grinschgl [Gri+12].

The DUP method only requires the information from the *golden run* to calculate the FESs based on the instructions' *write accesses* (write → define a data) and *read accesses* (read → use a data). Figure 2.10 shows the original FS from Figure 2.7 extended by data accesses from the golden run and the DUP. The x-axis represents nine system states due to eight executed instructions. The program run executes eight instructions until it terminates and reaches the final system state at t_8. The y-axis represents the total of four bits available and used by the program.

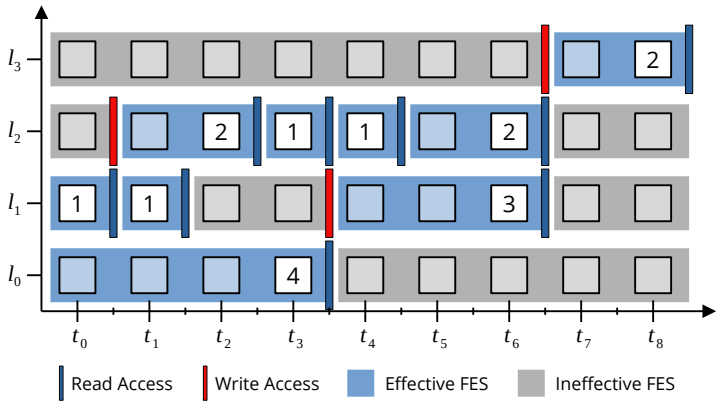


Figure 2.10 – Def/Use Pruning to Reduce the Number of Fault-Injection Experiments. DUP analyzes the golden run to determine between which of the nine system states (t_0 to t_8) the executed instructions perform write and read accesses on the four available bits (l_0 to l_3). Points of the FS are grouped into FESs considering this information. If an FES concludes with a write access, all FIs within the FES are considered benign. If an FES concludes with a read access, DUP selects a single pilot for the FES, and it maps the FI result to all other injections within the FES. The pilot is weighted based on the size of the FES to provide a statement about the complete FS.

With nine injectable system states over time and 4 bits, the total number of possible injections is $9 \times 4 = 36$. As previously mentioned, DUP considers both write and read accesses from the golden run. For instance, in the first instruction (the transition from t_0 to t_1), the execution reads bit 1 and writes the result of the instruction to bit 2, which is analogous to other instructions.[14] Data leaving the program scope (after t_8 at bit 3), such as when read by another process or via the IO port, is considered read access in DUP, which makes errors visible outside the SUT as unexpected, erroneous system behavior.

[14]For specific instructions, such as those from t_5 to t_6 or from t_7 to t_8, no explicit write or read accesses exist because they represent stalls or no-operations, for instance.

The basic principle of DUP is straightforward:

- All potential FIs of the entire FS are grouped into FESs on the temporal dimension (x-axis). The program start, program end, and each write or read access define FES boundaries.
- If an FES ends with a write access, it represents a part of the *ineffective* FS.
- If an FES ends with a read access, it represents a part of the *effective* FS.

The term *ineffective FS* means that FIs does not affect the SUT behavior. These FI are either overwritten by write access or not read again after reaching the final system state. Therefore, all FESs ending with a write access or the final system state are considered ineffective. These FESs represent the entire ineffective FS $\mathscr{I}$. This FS is ineffective, and all resulting FIs are benign. In the example in Figure 2.10, 18 FIs are ineffective. This simple consideration easily saves half of the FIs.

The *effective FESs* must be considered individually and cannot be grouped without further analysis. For each effective FES, designating a single *representative pilot* for the FI campaign is sufficient since the same read access handles all potential FIs, and it is irrelevant which of the FIs within the FES becomes the representative pilot. In this dissertation, the representative pilot designation is *inject-on-read* [Bar+05], meaning that the temporally latest possible FI is the pilot of the FES. Thus, the number of pilots equals the number of effective FESs, reducing the total number of FIs.

Since each pilot p_i of the n pilots overall is a representative whose FI result is mapped to all other FIs within the FESi, the pilots must be weighted to guarantee a complete statement about the considered FS. The weight function w determines the size of the respective FESi for the pilot's weight $w(p_i)$. DUP creates pilots that make a *complete* statement regarding its set of all possible FIs or, instead, the whole FS $\mathscr{F}$ (see Section 2.2.1). The number of *all* potential FIs $|\mathscr{F}|$ compared to the sum of all *weights* of the *pilots* $w(p_0), \ldots, w(p_{n-1})$ and the number of *benign* FIs of the ineffective FS $|\mathscr{I}|$ is the same:

$$|\mathscr{F}| = \sum_{i=0}^{n-1} w(p_i) + |\mathscr{I}|$$

After applying DUP in the example shown in Figure 2.10, the FI campaign only needs 9 FIs, a significant reduction compared to the 18 FIs required for the effective FS and an even more significant reduction from the 36 FIs required in the naive way, which means a 75 percent reduction of the FIs to make a complete statement about the FS. Barbosa and colleagues reported in their work [Bar+05] that in their real-world example of a jet-engine controller, the number of FIs in registers was reduced by two orders of magnitude, and for FIs in memory, up to five orders of magnitude reduction. Furthermore, the golden run processing, consideration of write and read accesses, and the determination of the DUP pilots are *deterministic*. Therefore, the subsequent FI-campaign results after pruning with DUP are always *precise*.

2.3.2.2 Heuristical Fault-Space Pruning

Although DUP significantly reduces the number of FIs required for *complete* coverage of the FS, the number can still be high, leading to a long runtime for the FI campaign. Several papers explore the further reduction of FIs using *heuristics*. I introduce an overview of some existing selected heuristic methods.

For example, Hari and colleagues developed the tool *Relyzer* [Har+12a; Har+12b; Har+13], and Venkatagiri and coworkers extended it for approximate computing, calling it *Approxilyzer* [Ven+16; Ven+19]. These tools employ heuristics based on complex considerations of control and data flows.

Their objective is to group existing FESs (e.g., FESs resulting from the DUP would be possible) into sets of FESs. One pilot represents the set of FESs, and its FI result is then applied to the entire set. Although Hari and colleagues report an accuracy of up to 94 percent, they do not address what to do if the accuracy is too low. If the FIs needed for the entire FI campaign are still too high, they recommend *sampling* the FI, which no longer covers the complete FS. Additionally, *Relyzer* focuses only on FI results classified as *benign* and SDC, neglecting other types of results like traps or invalid memory accesses that are becoming more relevant [DSE13; TP13]. Another tool, *SmartInjector* [LT13], uses similar heuristics to form sets of FESs, managing to reduce the number of FIs even further than Relyzer but with a focus solely on SDC. Schirmeier and colleagues provide another way to form sets of FESs based on the current system state at the time of the FI. However, instead of creating equivalences between FIs, the authors introduce the concept of *fault similarity* and pass FIs if a *similar* system state was injected before, achieving an accuracy of up to 99.84 percent [SBS14].

Lastly, *statistical fault injection* [Ram+08; Lev+09] involves randomly selecting a *sample* of FIs from the entire FS. This set of FIs is executed and assumed to be representative of the entire FS. The sample size depends on the result distribution and when this distribution meets a predefined *confidence interval*. This approach provides only a rough indication of the system's reliability. Sampling a handful of FIs from the entire FS and evaluating them to a specific confidence interval does not completely or precisely cover the FS.

2.3.3 Dynamic Fault-Injection–Experiment Acceleration

Another approach to accelerate an FI campaign is to accelerate individual FI experiments *dynamically* during their execution. The runtime of an FI experiment consists of two phases:

Pre-FI Phase This phase spans from the FI experiment's start until the actual FI performed.

Post-FI Phase This phase begins immediately after the FI and continues until the end of the FI experiment when the FI result becomes externally observable.

By targeting these two phases, it may be possible to reduce the overall runtime of the FI campaign. This dynamic acceleration can be applied per experiment, allowing for flexibility and efficiency in resource utilization.

Fast-forwarding techniques [FSK98; RSR99; Bar+05; Fid+06; SBK10; HK12] accelerate the system's execution by skipping portions of code until it reaches a specific dynamic instance of a static instruction, which means in the context of FI until the FI experiment reaches the pilot's injection time; in general, fast-forwarding techniques are not FI-specific. Schirmeier and colleagues developed a FI-specific fast-forward method, *smart-hopping* [SRS14], to accelerate the execution of FI experiments by utilizing the anyway extracted golden run and improving the FI experiment throughput by up to several magnitudes compared to similar approaches.

Checkpointing is a widely used technique to reduce the time from the start of an experiment to the injection [Par+00a; Par+00b; Ber+02; SH+14; Par+14]. This method involves storing system states at specific or equidistant time intervals. These stored states serve as *checkpoints* and are loadable anytime during an experiment. The pilot specifies the injection time, and if checkpointing is employed, the system loads the checkpoint closest to the injection time. Consequently, the time between the experiment's start and the checkpoint does not need to be executed since the system state is already available. However, it is essential to consider that loading and saving checkpoints incur additional time and memory costs [Ber+02]. Therefore, the decision to use checkpoints should be made judiciously, considering the associated overheads.

Another approach to accelerate result generation is to *monitor* the experiment post-FI and *predict* the impact of the FI on the SUT's behavior. This strategy can lead to early termination of the experiment, thereby saving overall runtime. One developed approach for lower system layers is *dynamic fault collapsing* [Ber+02]. After injection, this method regularly compares the system state of the SUT with that of the golden run. This method maps differences detected to the original fault. If the resulting system behavior of the differing bit is known, the FI experiment terminates, and the method uses the result of the differing bit. If the result is not yet known, the experiment is completed regularly, and one more differing bit is known for the next experiment.

The *SmartInjector* [LT13] operates on a similar principle at the ISA layer, focusing on benign faults or that lead to an SDC. Instead of comparing the whole system state, this method compares system outputs or the behavior of statically determined masking, outputs, or branch instructions to the golden run.

The approach *GangES* [SH+14] also groups FI experiments into "gangs", resulting in the same intermediate machine state, so that only one gang simulation is necessary. It leverages program structures to determine when and which states to compare with the golden run, building on a concept similar to dynamic fault collapsing.

2.3.4 Summary of Accelerating Fault-Injection Campaigns

In order to make a comprehensive statement about the susceptibility of faults across the entire FS, it is not feasible to naively execute each FI in the FS one by one without any optimizations. The example involving the ARM Cortex-M33FU5 and the anti-lock braking system illustrates the potential *fault-injection–campaign runtime explosion*, indicating that it would take at least 1.2 years to cover the FS cycle-wise *completely*. To address this challenge, two primary approaches to accelerate an FI campaign are reducing the required FIs and decreasing the average FI-experiment runtime.

Several *pruning* methods exist to reduce the number of FI experiments. *Def/Use Pruning (DUP)* is a precise method that groups FIs into FESs, requiring only one FI (the *representative pilot*) per FES. Additionally, heuristic pruning methods use program execution knowledge to decrease the number of FIs needed further, often resulting in sets of FES with a single pilot. Although these methods are generally *imprecise*, their deviation is acceptable or context-dependent.

To save time during the execution of a single FI experiment, techniques such as *checkpointing* can be employed, allowing the system to revert to a specific point before injection. Another approach to expedite FI experiments is to *predict* results early in the post-injection runtime.

This dissertation introduces three contributions for accelerating FI campaigns: Two new pruning methods for further reducing the number of FI experiments in a campaign and one dynamic outcome predictor for reducing the FI experiment runtime during execution.

- In Chapter 4, I present *Data-Flow–Sensitive Fault-Space Pruning* [▷PDL21], which consider not only the temporal aspect but also the *semantics* of individual instructions to form *precise* sets of FES and cover the FS *completely*.
- In Chapter 5, I introduce *Program-Structure–Guided Fault-Space Pruning* [▷Pus+19], which utilize *program structures* determined from the golden trace to calculate *Fault-Space Regions* within the FS, which again determine sets of FES, with only one FI executed. Although this method is heuristic, it exhibits a slight deviation, making it practical and still *complete*.
- Finally, in Chapter 6, I present *ACTOR* [▷Tho+22], a *dynamic outcome prediction* method that deals with predicting *timeouts* during a running FI experiment. This heuristic method dynamically examines jump sequence correlation at runtime to improve prediction accuracy.

3

Implementation and Evaluation Process

> If you feel safe in the area you're working in, you're not working in the right area. Always go a little further into the water than you feel you're capable of being in. Go a little bit out of your depth. And when you don't feel that your feet are quite touching the bottom, you're just about in the right place to do something exciting.
>
> DAVID ROBERT JONES, BETTER KNOWN AS "DAVID BOWIE" (1947–2016)

This chapter provides an overview of the tools used and implemented within this dissertation, mainly focusing on the technical implementation of the FI-campaign process (refer to Section 2.2.2 on page 35). One of the primary components is the simulation-based FI framework, *FAIL**, a central framework used throughout this dissertation. As highlighted in the preceding chapter, the primary emphasis lies on FIs targeting the ISA layer, which aligns with the principal focus of FAIL*'s FI capabilities. This framework specializes in injecting bit flips on the ISA layer, tracing error propagation, and simulating resulting behavior or, more specifically, generating a *failure classification* derived from a bit flip.

The contributions of this dissertation *expand* the functionality of the existing FI framework FAIL*, aiming to accelerate the execution of FI campaigns conducted with FAIL*. I introduce an essential concept to comprehensively evaluate and quantify the quality of these FI campaign acceleration methods: the notion of the *effectiveness* of FI-campaign acceleration. To facilitate a quantitative assessment of the effectiveness of the contributions made in this dissertation, I outline the process of determining an FI campaign reference execution – the *ground truth* and elaborate on the evaluation

methodology for assessing these contributions against the established ground truth. Lastly, I introduce the *benchmark portfolio* used in this work and the *technical setup* adopted for conducting the research.

3.1 The Fault-Injection Framework FAIL*

The simulation-based FI framework FAIL* plays a central role in the research conducted in this work. This section provides an overview of FAIL*, outlining its general workflow and highlighting the specific components employed in this dissertation.

FAIL* operates by executing simulations of the SUT and intentionally injecting faults to observe the system's response. The workflow involves distinct stages, from initializing the simulation to injecting faults and subsequent recording of the outcomes.

One crucial aspect of FAIL* is its ability to generate diverse FI-experiments results. These results are categorized into different classes of failures, each representing a distinct manifestation of system vulnerability.

A essential contribution of this dissertation is the integration of novel methodologies within the FAIL* framework. Specific enhancements and optimizations are implemented at strategic points in the workflow to improve the efficiency and effectiveness of FI experiments. The upcoming chapters 4 - 6 will provide a detailed account of these contributions and their integration within the FAIL* framework.

3.1.1 FAIL* Procedure

FAIL*[15] [Sch+15; Sch16] is designed to inject faults during the runtime of an executed program on simulated fixed hardware. It leverages various backend simulators, enabling the simulation of diverse hardware models. This flexibility allows FAIL* to execute programs on simulated hardware and dynamically inject faults at runtime, facilitating a comprehensive evaluation of the system's reliability.

The FAIL* supports the simulation and injection of faults on different processors, including an x86 processor using the Bochs simulator,[16] an ARM A9 processor with the Gem5 simulation,[17] and QEMU.[18] Additionally, it works with the OpenOCD debugging interface,[19] enhancing its versatility and applicability across various hardware architectures.

FAIL* adopts a client-server architecture for its implementation. The *job server* is essential, housing essential information for FI campaign execution. This server delegates individual FI experiments from the campaign to diverse *clients*, transmitting corresponding *pilots*. Each client executes an individual FI experiment using the provided pilot information in the simulated hardware. Subsequently, the client reports the observed SUT behavior back to the server.

At the core of FAIL* lies a database that encompasses all requisite data for running an FI campaign and recording its results. FAIL* is flexible, allowing users to craft customized FI campaigns and integrate specific tools and plugins to enhance and tailor the campaigns according to their needs.

Executing a campaign and obtaining results about the reliability of the evaluated system using FAIL* involves completing the four FAIL* *assessment cycle* steps, as depicted in Figure 3.1 [Sch+15; Sch16]. This assessment cycle is a specific realization of the FI-campaign process outlined in Section 2.2.2 on page 35. The individual steps of the assessment cycle are as follows:

[15] *"FAIL*- short for FAult Injection Leveraged, with the asterisk highlighting its variability regarding target backends"* [Sch16]. FAIL*'s source code is available on GitHub at the following link: `https://github.com/danceos/fail`.

[16] *"Bochs is a highly portable open source IA-32 (x86) PC emulator written in C++, that runs on most popular platforms. It includes emulation of the Intel x86 CPU, common I/O devices, and a custom BIOS."* – `https://bochs.sourceforgie.io/`

[17] *"The gem5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture and processor microarchitecture. gem5 is a community-led project with an open governance model."* – `https://www.gem5.org/`

[18] *"A generic and open source machine emulator and virtualizer"* – `https://www.qemu.org/`

[19] *"OpenOCD, the Open On-Chip Debugger has been created by Dominic Rath as part of a diploma thesis at the University of Applied Sciences, FH-Augsburg."* – `https://openocd.org/`

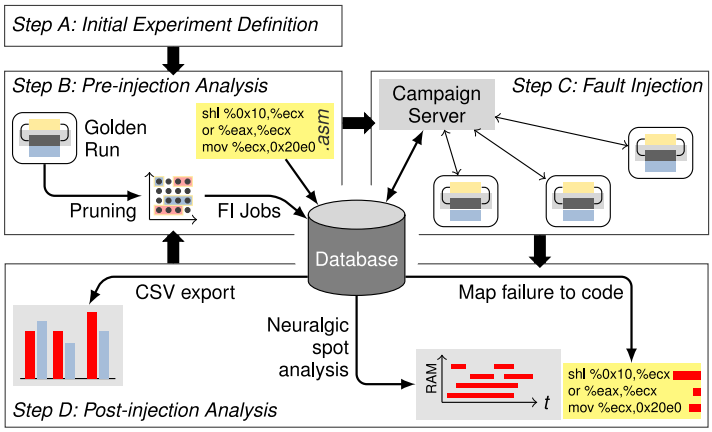


Figure 3.1 – FAIL* Assessment Cycle.
Following the (A) initial definition, the (B) SUT undergoes analysis using the golden run, supplying essential information to the central database regarding the SUT and the intended FIs. Subsequently, (C) the campaign is executed, and (D) the results are subject to analysis. This iterative process allows for extracting new insights from the SUT's resulting behaviors, contributing to the refinement of subsequent FI campaigns. The SUT undergoes iterations until it achieves the desired level of understanding or system hardening. *The figure is taken from [Sch+15].*

Step A - Initial Campaign Definition To conduct an FI campaign, the initial step is to define a *FAIL* campaign*.[20] This step specifies the campaign type and its parameters through predefined or self-developed FAIL* templates. In this phase, FAIL* does not execute any FIs. I limit the use of FAIL* to the following two generic FAIL* campaign templates, applied in the listed order:

1. To analyze the effects of bit flips in the SUT, the framework requires a reference execution of the program, known as the *golden run* (see Section 2.2.2 on page 35). The FAIL* template *generic tracing* creates the golden run using the built-in plugin *tracing*. Upon completion, the golden run is prepared for use in the subsequent FAIL* step.
2. The *generic (FAIL*) experiment* generates the pilots for the planned FI campaign. This step operates through the job server and is executed within the *client-server architecture* using the golden run. This server coordinates the distribution of pilots to numerous clients for *parallel* execution and subsequently records the results in the central database.

Step B - Pre-Injection Analysis Following the definition of the campaign, the *pre-injection–analysis* step is initiated. Depending on the FAIL* campaign definition, this phase involves *creating* or *analyzing* the *golden run*. When analyzing the golden run, IP and memory events are read (as shown in Listing 2.1 on page 36), and FAIL* writes the data to its central database. Importing the *objdump* of the compiled programs of the SUT can enhance the human interpretability of individual IPs if desired. Subsequently, FAIL* calculates the pilots for the upcoming FIs based on the golden run, incorporating acceleration techniques like DUP (see Section 2.3.2.1 on

[20] A FAIL* campaign is termed by Schirmeier [Sch+15] a FAIL* experiment, which includes both the definition and execution of an FI campaign, which comprises individual FI experiments. In the context of this dissertation and my contributed papers, the terms *FAIL* experiment*, *FAIL* campaign*, and *FI campaign* are used interchangeably to especially separate the overall FI campaign and single FI experiments on the SUT.

page 47), the default pilot calculation technique in FAIL*. As this step concludes, the pilots are readily available in the database for the job server in the subsequent step.

This step will be the recurring focus throughout the dissertation; in this pre-injection step, I implemented the first two of my three contributions to reducing the number of FI experiments in the FAIL* campaign.

Step C - Fault Injection In this step, the planned FI campaign is executed. The *job server* retrieves the *pilots* from the database and assigns the associated workload to the *clients*. Each pilot triggers the creation of a *breakpoint* when executing the SUT on the client, specifying the time of the break and the bit to be flipped. At this breakpoint, the client executes the code depicted in Listing 3.1. The breakpoint predetermines the injection time, and the injection location is passed through the code in the function with the signature `DatabaseExperiment::injectionFault`. Using the parameters `data_address` and `bit_position`, the client precisely identifies the bit to be flipped. Accessing the registers in the SUT requires a separate disassembler. Currently, FAIL* supports the LLVM[21] and Capstone[22] disassemblers; I use Capstone in the context of this dissertation.

Listing 3.1 – FAIL*'s Code of a Fault Injection.
`DatabaseExperiment::injectionFault` accepts the parameters `data_address` and `bit_position`, defining the bit in the SUT to be flipped in the FI experiment. Due to Bochs-related considerations, byte data addresses below address `2048` serve as a mapping for the general-purpose register. FAIL* employs a disassembler (`XtoFailTranslator`) for register injections that reads and writes values from and to specific registers. FAIL* uses an internal memory manager (variable `m_mm`) in the case of FI into memory. *This listing contains the original FAIL* code from the `src/core/ewf/DatabaseExperiment.cpp` file, albeit simplified for clarity in explanation.* The highlighted lines show the actual bit flip.

```
1  unsigned DatabaseExperiment::injectFault(
2      address_t data_address, unsigned bitpos, bool inject_burst,
3      bool inject_registers, bool force_registers, bool randomjump) {
4
5      unsigned value, injected_value;
6      // FI into a general purpose register
7      if (data_address < 2048 && inject_registers) {
8          // decode the related register and get its value
9          XtoFailTranslator::reginfo_t reginfo =
10             XtoFailTranslator::reginfo_t::fromDataAddress(data_address, 1);
11         value = XtoFailTranslator::getRegisterContent(simulator.getCPU(0), reginfo);
12
13         // inject the fault into the value
14         injected_value = value ^ (1 << bitpos);
15         // write the injected value back to the register
16         XtoFailTranslator::setRegisterContent(simulator.getCPU(0), reginfo, injected_value);
17
18     // FI into memory
19     } else {
20         // get the value via FAIL*'s MemoryManager
21         // from the memory at 'data_address'
22         value = m_mm.getByte(data_address);
23
24         // inject the fault into the value
25         injected_value = value ^ (1 << bitpos);
26         // write the injected value back to the memory
27         m_mm.setByte(data_address, injected_value);
28     }
29 }
```

[21] *"The LLVM Project is a collection of modular and reusable compiler and toolchain technologies."* – `https://llvm.org/`

[22] *"Capstone is a lightweight multi-platform, multi-architecture disassembly framework. Our target is to make Capstone the ultimate disassembly engine for binary analysis and reversing in the security community."* – `https://capstone-engine.org/`

Lines 14 and 25 illustrate the concrete bit flips in the code using the `XOR` (`^`) operation. Due to the isolated execution of an FI experiment within the backend simulation of a client and the independence of all FI experiments, this process is amenable to parallelization. Upon completion, the client reports the resulting system behavior after the FI to the job server. After this stage, a *result table* will be placed in the database, encompassing outcomes for each pilot in the entire FI campaign.

In this step, I have implemented the last of my three contributions. During the campaign runtime, this improvement in FAIL* accelerates the execution of individual and specific FI experiments.

Step D - Post-Injection Analysis Following the completion of the FI campaign and the retrieval of the trace data and result table, the subsequent step is the *post-injection analysis* of the entire campaign results. Leveraging the imported trace data and the database structure of FAIL* enables the quantification and analysis of the SUT's reliability. Precisely tracing back the FI behavior in the executed code of the SUT allows for informed conclusions regarding potential improvements and opportunities for hardening the SUT. Armed with this enhanced understanding or an improved, fortified version of the SUT, the cycle can recommence at step B. This iterative process, the *FAIL* assessment cycle*, spans multiple FI campaigns, continuously refining and enhancing the SUT.

Regarding the eight criteria for evaluating FI techniques (refer to Table 2.1 on page 41), FAIL*, being a simulation-based FI framework, excels in several aspects: (1) *repeatable*, ensuring deterministically reproducible results, (2) providing complete *controllability* through the simulation, (3) offering *observability* over the SUT execution and the effects of FIs, (4) being *non-intrusive* as FAIL* injects the program semantically unchanged, (5) ensuring *reachability* at every point in the ISA layer FS, (6) enabling *scalability* through arbitrary parallelization, and (7) being *cost-effective* by avoiding the need for expensive hardware prototypes. The accuracy of FI campaign results depends on the chosen backend simulator and its implemented (8) *degree of realism*.

Throughout this dissertation, I employ FAIL* for all FIs and enhance its functionality with new features implemented as FAIL* plugins and FAIL* tools. Given that the Bochs simulator is a well-established x86 processor simulator, my work consistently refers to the Bochs simulator and the simulated x86 processor in the backend simulation for FAIL*.

3.1.2 Failure Classification

FAIL* *observes* various types of *system behaviors* following the execution of an FI as the injected fault propagates and traverses the *system boundary* of the SUT (refer to Section 2.1.3 on page 19). If the SUT's behavior after an FI matches the behavior without an FI (i.e., the golden run), the fault is labeled *benign*; otherwise, FAIL* recognizes it as an *observable system failure*.

The crucial aspect is not whether the execution or system states of the SUT align with the golden run in any manner but rather the observable behavior *outside* the system boundary after the completion of the execution. As I focus on classifying *system failures* rather than individual errors within the system, the system state itself is not a decisive factor. Only the resulting system behavior after the termination of the SUT is pertinent.

To this end, the following *(system) failure classes* are defined and considered within the context of FAIL* and this dissertation:

Benign This class refers to cases where the execution of the SUT terminates, and it behaves as anticipated, yielding the expected result. Injected executions free of failures can occur by implementing error detection methods with appropriate responses or successful error correction methods. The possibility of masking bit flips (as seen in the binary `AND` instruction, where a bit flip at one input becomes benign due to the other input bit being a constant *zero*) could also result in externally observed failure-free SUT execution. As emphasized earlier, the system state is inconsequential for this failure class; the focus lies solely on the externally observable failure-free system behavior.

Silent Data Corruption *Silent Data Corruption (SDC)* signifies a failure class wherein the SUT terminates its execution and appears to behave as anticipated following an FI. However, if the expected result is known, the result returned after termination is *incorrect*, leading to a potentially observable erroneous output. The severity of this failure class becomes critical, especially when it is challenging to determine an expected result (e.g., calculations based on sensor data in a non-reproducible real-world environment like radiation). Thus, SDC is the *most dangerous* failure class because, in the worst case, this type of system failure allows the system to continue its execution as if it would process correctly.

Timeout A timeout refers to an execution of the SUT that does *not* terminate within a specified time. If injected data alter the control flow, deviations from the golden-run's control flow are possible. In principle, this is not serious, as the system can still produce the correct and expected result. However, a *timeout* occurs if the control flow does not exit for an extended period.

For instance, if an FI flips a bit of a loop variable, it can lead to an infinite loop, preventing the system from terminating. It may also be unclear whether the SUT will ever terminate, known as the *Halting Problem* [Gö31; Chu36; Tur37]. Therefore, a very high *threshold* is usually used as an approximation, which is a multiple of the execution time of the golden run. If the SUT runs longer than this threshold, it is a timeout with a high probability. For safety-critical systems, typically real-time systems, if the execution exceeds the specified threshold, it violates the specification and is considered insufficiently safe.

Trap A trap describes a (software-side) synchronous interrupt.[23] This failure class does not address any hardware-side or asynchronous interrupts. A prominent example of a trap is dividing a value by zero. This failure class also includes invalid memory accesses, such as writing outside the allocated memory space or writing to the text segment of the program, which can lead to loading corrupted instructions.[24]

The framework can identify additional system failures or more nuanced failure classes beyond those mentioned above, which proves helpful in specific specialized experiments. However, the specified failure classes suffice for the contributions in this dissertation. This selection of failure classes enables a precise classification of all FIs across the entire FS.

[23] As mentioned in Section 3.1, I use the Bochs simulator as the FAIL* backend hardware model in the context of this dissertation. Throughout the dissertation, the term *trap* is potentially used for any software-side interrupt, which, in the x86 ISA, is generally referred to as an *exception* [Cor22]. However, distinguishing the terms of different interrupts is not straightforward because *"no clear consensus as to the exact meaning of these terms"* exists over several different ISAs [Hyd10]. Therefore, I will use the term *trap* to refer to any software-side interrupt.

[24] In the context of Bochs in FAIL*, serving as an x86 simulator, I assume that, similar to the x86 architecture, a memory management unit detects invalid memory accesses and triggers a trap in the system.

3.1.3 FAIL* Plugins and Tools

FAIL* offers extensibility by incorporating FAIL* plugins and FAIL* tools. During the initial campaign definition in step A of the assessment cycle, FI campaign designers can configure the necessary compilation steps, allowing pre-configuration of FAIL* plugins.

FAIL* plugins provide modular functionality typically required during the execution of a single FI experiment. For example, the *Tracing* plugin, used by FAIL*'s *generic tracing*, records the instruction and memory-access trace of the workload in a compressed file. This trace file serves as the golden run for calculating pilots in the pre-injection–analysis step of FAIL*'s *generic (FAIL*) experiment* [Sch+15; Sch16]. Notably, the development of the *autocorrelation* plugin, which collects runtime data during a single FI to determine potential timeouts using an autocorrelation algorithm, contributed to this work. I will explain this approach in more detail in Chapter 6.

The second option for extending FAIL* is implementing or enhancing FAIL* tools. These tools operate independently of the FI campaign and serve diverse purposes. For example, after successfully creating the golden run, the *dump-trace* tool makes it more human-readable. The *import-trace* tool facilitates the import of the golden run into the FAIL* database, supporting various formats. Additionally, the *prune-trace* tool, extended in the context of this dissertation, offers options to use Data-Flow Pruning (more in detail in Chapter 4) or Fault-Space Regions (more in detail in Chapter 5). These FAIL* tools, though related to FI-campaign definitions, can be used independently for analyses without executing any FIs.

FAIL* tools primarily play a role in the pre- and post-injection analysis steps, although FAIL* plugins are more involved in the fault-injection step, except for tracing. However, both may be present in every phase of the assessment cycle to some extent. Figure 3.2 illustrates the FAIL* assessment cycle, as seen in Figure 3.1, with highlighted sections representing the implemented contributions of this dissertation. These implementations aim to enhance the feasibility of FI campaigns by providing a complete assessment of the FS and the system's reliability when executing software on fault-prone, fixed hardware.

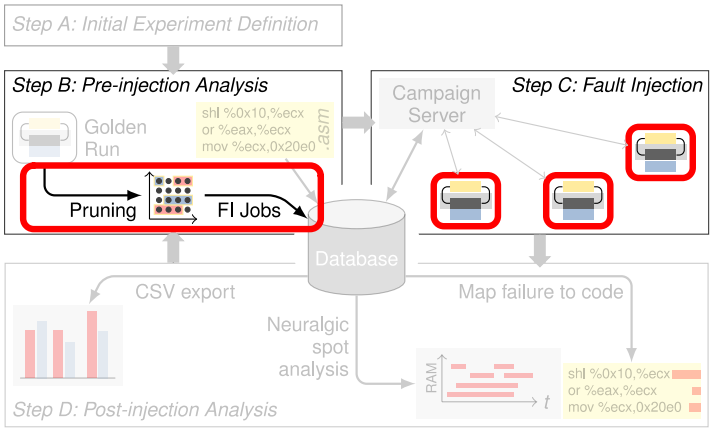


Figure 3.2 – Focus of this Dissertation Regarding the FAIL*'s Assessment Cycle.
This dissertation centers on diminishing the overall execution time of FI campaigns. To attain this objective, two pruning methods and one monitoring method within FAIL* are developed and implemented. Subsequent chapters detail the enhancements made to FAIL* by integrating new FAIL* plugins and tools. *The figure is adapted from [Sch+15].*

The code developed during this dissertation is publicly accessible in the FAIL* GitHub repository and available on Zenodo.[25]

3.2 Evaluating Fault-Injection–Campaign Improvements

As the primary objective of this dissertation is to enhance the overall runtime of FI campaigns, I outlined in Section 2.3.2 on page 46 and Section 2.3.3 on page 49 specific methods to decrease either the number of pilots n or the FI-experiment runtime t_{exp}. To evaluate the effectiveness of these FI-campaign accelerating methods, I define the notions of *effectiveness* and *efficiency* within the context of FI acceleration methods and describe the evaluation process using FAIL*. Following this, I elaborate on the benchmarks used in this work to measure the effectiveness of the contributed FI-campaign acceleration methods and the technical setup employed to collect these results.

3.2.1 Fault-Injection–Campaign Acceleration Effectiveness and Efficiency

The precise definitions of *effectiveness* and *efficiency* are crucial to understanding the contributions of this work. In general, the definitions of these terms are as follows:

Effectiveness Effectiveness is a goal-dependent criterion that assesses the difference between the desired and the achieved goals. A more comprehensive understanding of effectiveness always relates to outcome measures [Nul98]. The nearer the desired and archived goals are, the higher the effectiveness.

Efficiency Efficiency is an explicitly goal-independent criterion based on the quantitative correlation between the achieved goal and the resources expended to accomplish it [Nul98]. It is a higher level of efficiency when fewer resources are required.

The *goal* of an FI framework is to provide comprehensive results of all individual FIs of a campaign to assess the SUT's reliability; in the context of this dissertation, the goal also concludes, aiming for FS-completeness. Regarding accelerating FI campaigns, it is essential to differentiate between the FI framework in use and the independent FI-campaign acceleration methods in terms of efficiency and effectiveness. A more detailed explanation of these distinctions will follow in the next two sections.

3.2.1.1 Ground Truth

A reference for comparison is needed to assess the performance of an acceleration method. This reference – the *ground truth* – serves as a basis and must represent a particular standard of execution of an FI campaign, the golden run.

Acceleration methods influence the number of experiments n or the individual experiment runtimes $t_{\text{exp},i}$ to improve the overall campaign runtime t_{cpn}. These values for both need to be determined to assess the performance of the acceleration methods. Additionally, in the case of evaluating approximative methods that approximate observed system behavior for specific points in the FS or predict system behavior during experiment execution, the observed and recorded system behaviors from a non-approximative reference execution are necessary to determine the correctness of the acceleration methods under evaluation.

The ground truth in the context of this dissertation consequently contains three major parts:

[25] *"Zenodo is a research data repository and publishing platform that enables researchers to store, share, and preserve their research outputs. It provides a persistent and citable digital identifier for datasets, software, publications, and other research artifacts, making them easily discoverable and accessible to the wider scientific community."* – `https://zenodo.org/`

Number of Pilots The naive approach to determining pilots for a campaign entails defining a pilot for every accessed time point and location captured in the golden run, which is the total number of potential FIs in the entire FS $n_{\text{naive}} = |\mathscr{F}|$. However, employing this vast number of pilots in the FI campaign proves infeasible (refer to Section 2.3.1 on page 45).

The *Def/Use Pruning (DUP)* (see Section 2.3.2.1 on page 47) is a well-known and popular standard method because of its fast computation speed, precision, FS-completeness, and relative ease of implementation [Smi+95; GS95; Ben+98b; Ber+02; BP03; Bar+05; Gri+12], that reduces the number of pilots significantly [Bar+05]. Since DUP is a fundamental method and today's basis for novel acceleration methods in the research community (as it is also in two of my three contributions), I consider the results of this method as the ground truth for measuring the performance of newly developed acceleration methods optimizing the number of pilots.

$$\text{ground truth}_n \coloneqq n_{\text{DUP}} = |P_{\text{DUP}}|$$

Accordingly, the set of DUP pilots P_{DUP} or its number of pilots n_{DUP} represents this part of the ground truth. The pilots from P_{DUP} are also the basis for this list's two subsequent parts.

Individual Experiment Runtimes In its simplest form, the FI-experiment runtime is the duration from the execution of the first instruction in the experiment to the final report of the observed system behavior post-FI. The ground truth must contain the recorded runtimes of all experiments $t_{\text{exp},i}$ to accurately measure the impact of acceleration methods on campaign time during experiment execution. Here, as the set of the ground truth's pilots is P_{DUP}, only the experiment runtimes considering these pilots belong to the determination of all experiment runtimes $T_{P_{\text{DUP}}}$. Thus, this part of the ground truth is as follows:

$$\text{ground truth}_{\min t_{\text{exp}}} \coloneqq T_{P_{\text{DUP}}} = \bigcup_{i=0}^{n_{\text{DUP}}-1} t_{\text{exp},i}$$

No (non-default) acceleration methods must be active during every experiment execution, ensuring that individual runtimes remain primarily unaffected and comparable. The sum of all elements in $T_{P_{\text{DUP}}}$ equals the overall campaign runtime t_{cpn}.

Campaign Results After concluding the campaign, all FI-induced system behaviors are observed and recorded. For every FI experiment i with the pilot p_i, the injected SUT results in a specific classified failure c_i, thereby defining a single *FI-experiment result* denoted as $r_i = (p_i, c_i)$. The collection of all FI-experiment results defines the overall *FI-campaign results* as the set R_{DUP}:

$$\text{ground truth}_R \coloneqq R_{\text{DUP}} = \bigcup_{i=0}^{n_{\text{DUP}}-1} r_i$$

Although R_{DUP} might not be directly relevant for measuring the campaign runtime, it is crucial to assess the precision of the *approximative* acceleration method using heuristic approaches.

3.2.1.2 Fault-Injection Framework Effectiveness and Efficiency

As an FI campaign aims to provide FI results for every point in the FS, FAIL* by default achieves maximal *effectiveness* in this regard and addresses every potential FI in the FS. However, the effectiveness is compromised if sampling or heuristic methods either incompletely or inaccurately determine FI results regarding the FS.

The *efficiency* of executing an FI campaign depends on two key factors: (1) Firstly, it depends on the implementation of the FI framework itself and its resulting *framework overhead*. I use FAIL* as the FI framework, and I consider the implementation details of the FI process and individual FI executions to be sufficiently efficient. Thus, this factor is different from the focus in this dissertation. (2) Secondly, the efficiency of the FI-framework execution is influenced by reducing the pure FI-campaign runtime t_{cpn} of the FI campaign. Consequently, the shorter t_{cpn} is, the more efficiently the framework's goal is reached. The reduction in t_{cpn} could make previously infeasible workloads of FI campaigns feasible, as discussed in the context of campaign runtime explosion in Section 2.3.1 on page 45.

I primarily focus on optimizing the FI-campaign runtime t_{cpn} to complete FS coverage *more efficiently*.

3.2.1.3 Fault-Injection Campaign Acceleration Effectiveness and Efficiency

FI-Campaign–runtime acceleration methods address one of two factors of optimizing $t_{\text{cpn}} = n \cdot t_{\text{exp}}$: either the reduction of FI experiments n in the pre-injection step or the reduction of t_{exp} during the FI-experiment execution. The theoretically ideal goal of both acceleration method types is to achieve $t_{\text{cpn, ideal}} = 0$. However, this ideal scenario is practically unrealistic, but acceleration methods aim to bring the actual campaign runtime t_{cpn} as close as possible to this ideal value $t_{\text{cpn, ideal}}$.

Therefore, the *effectiveness* of a specific acceleration method is determined by how much it reduces the runtime t_{cpn}, essentially minimizing n or minimizing t_{exp} to bring t_{cpn} closer to $t_{\text{cpn, ideal}}$. Conversely, the *efficiency* of acceleration methods refers to the resources needed to reduce either n or t_{exp}.

Effectiveness of Reducing the Number of Pilots

To evaluate the *effectiveness* of acceleration methods in reducing n, a *ground truth* must be defined to compare the resulting n.

The value n_{DUP} is the ground truth concerning the number of pilots required for full FS coverage. The effectiveness regarding $\min n$ is quantifiable with DUP's pilots as the ground truth.

Let us assume a new acceleration *method* that calculates n_{method} pilots, covering the FS entirely. The calculation of the new method's effectiveness compared to DUP is as follows:

$$\text{Effectiveness}_{\min n} := \frac{n_{\text{DUP}} - n_{\text{method}}}{n_{\text{DUP}}} \tag{3.1}$$

This formula represents the ratio reduction of the new method's number of pilots compared to DUP, providing a way to gauge the new method's effectiveness.[26] Throughout this work, I will refer to this definition of effectiveness and the reduction in the percentage of experiments.

[26] Theoretically, if the number of the new method's pilots exceeds that of the DUP, the effectiveness becomes negative by definition. The negative result implies that the new method is useless in every sense, which should already be apparent when solely comparing the absolute numbers of pilots; a new acceleration method should always guarantee $n_{\text{method}} \leq n_{\text{DUP}}$.

Effectiveness of Reducing the Experiment Runtime

To evaluate the effectiveness of acceleration methods that target $\min t_{\text{exp}}$, we need to focus on specific parts of the entire FI-campaign runtime. For the evaluation, the pilots must be the same for each assessment.

Let P represent the set of pilots determined in the pre-injection–analysis step (e.g., using DUP). Each pilot $p_i \in P$ denotes an FI at a specific time and location in the SUT, requiring a corresponding time $t_{\text{exp},i} \in T_P$. Here, T_P contains all runtimes of individual FI experiments initiated by each pilot p_i.

The overall campaign duration t_{cpn} is as follows:

$$t_{\text{cpn}} = \sum_{i=0}^{|T_P|-1} t_{\text{exp},i} + t_{\text{ov}_{\text{framework}}}$$

Whereas the framework overhead $t_{\text{ov}_{\text{framework}}}$ is an inherent part of the FI-campaign runtime, its optimization is beyond the scope of this dissertation. This work focuses on optimizing FI campaigns through acceleration methods rather than directly refining the existing FI framework (in my case FAIL*).[27]

The runtime of the i-th individual FI experiment $t_{\text{exp},i}$ comprises two primary components: (1) Firstly, it includes the pure program runtime $t_{\text{prg},i}$ which signifies the duration an FI experiment runs on the SUT until its completion. This runtime extends until the FI framework adequately classifies the FI's impact, assigning it to one of the failure classes, as discussed in Section 3.1.2. (2) Secondly, the overhead associated with each experiment has two distinct parts. The first part is the overhead induced by the FI framework. The framework overhead summarizes the necessary computational resources required by the FI framework to execute and manage all individual FI experiments of the campaign. As mentioned before, this work does not focus on optimizing the framework overhead. The second part is any overhead conditioned by the new acceleration method, denoted as $t_{\text{ov}_{\text{method}},i}$.

By combining these parts, the overall FI-experiment runtime is:

$$t_{\text{exp},i} = t_{\text{prg},i} + t_{\text{ov}_{\text{method}},i}$$

Similar to the FI-campaign runtime, the framework-induced overhead *per experiment* is disregarded in this evaluation, as the primary focus is not to optimize the framework's overhead in any sense and, thus, will always be considered constant.

Nonetheless, reducing individual FI-experiment runtimes impacts the overall FI-campaign runtime directly, which involves minimizing the program runtime up to failure classification t_{prg} and the overhead caused by the acceleration method $t_{\text{ov}_{\text{method}}}$. Consequently, the considered FI-campaign runtime is defined as follows:

$$t_{\text{cpn}} = \sum_{i=0}^{|T_P|-1} t_{\text{exp},i} = \sum_{i=0}^{|T_P|-1} \left(t_{\text{prg},i} + t_{\text{ov}_{\text{method}},i}\right) \tag{3.2}$$

Defining a *ground truth* becomes essential once again to determine the effectiveness of an acceleration method considering $\min t_{\text{exp}}$. In this context, an FI framework operates by reducing the FI-experiment runtime by utilizing its implemented optimizations during the execution of the SUT.

[27] In the case of FAIL* and its client-server architecture, the evaluation environment's infrastructure overhead might be a factor in determining its technical implementation. However, this aspect does not significantly impact the core contributions of this dissertation. Therefore, the contributions of this work do not explicitly delve into optimizing or focusing on this specific infrastructure overhead within this work.

The FI framework executes the campaign plainly to establish the ground truth and avoid affecting the SUT's execution.

Assuming the development of a new acceleration method designed to optimize $\min t_{\text{exp}}$, the assessment of this method's effectiveness is a direct comparison between its FI-campaign runtime $t\text{cpn}_{\text{method}}$ and the standard execution of the unaffected and *plain* FI-experiment setup:

$$\text{Effectiveness}_{\min t_{\text{exp}}} := \frac{t_{\text{cpn}_{\text{plain}}} - t_{\text{cpn}_{\text{method}}}}{t_{\text{cpn}_{\text{plain}}}} = \frac{\sum_i t_{\text{exp}_{\text{plain}},i} - \sum_i t_{\text{exp}_{\text{method}},i}}{\sum_i t_{\text{exp}_{\text{plain}},i}}$$

The reduction in individual experiment runtimes $t_{\text{exp},i} \in T_P$ compared to the plain setup (i.e., the ground truth) directly determines the effectiveness. Even reducing a single experiment's runtime influences the effectiveness of the acceleration method. However, this reduction spans multiple or all n experiments, decreasing total FI-campaign runtime t_{cpn}. In that case, it implies a significant increase in the effectiveness of the acceleration method for optimizing $\min t_{\text{exp}}$.

Acceleration Method's Efficiency

The *efficiency* of any acceleration method describes how many resources (e.g., CPU time, memory) it requires to perform the method computation regarding $\min n$ or $\min t_{\text{exp}}$ compared to a process without any applied acceleration methods. The efficiency of the acceleration method depends on its concept and implementation. However, when assessing efficiency, it is essential to distinguish between $\min n$ and $\min t_{\text{exp}}$ and their impact on t_{cpn}.

The optimization of $\min n$ happens during the pre-injection–analysis step (i.e., before the actual execution of the FI campaign). The efficiency of such a method reflects how resource-intensive it is in calculating the pilots (e.g., CPU time needed, memory space). In practice, the resources used during the execution of these methods are not directly relevant for t_{cpn}. It would become relevant only if the method computes pilots for longer than the reduction through the pilots, which provides an advantage of the overall campaign runtime t_{cpn}. However, if the FS under evaluation is sufficiently large and the average experiment duration is long enough, the efficiency of a method computation $minn$ would not significantly affect the overall runtime savings from the very start of the FI process to the end of the last FI experiment.

Conversely, the efficiency of methods considering $\min t_{\text{exp}}$ directly influences the experiment's runtime and usually has an additional overhead t_{ov} (as seen in Equation 3.2); high overhead impacts t_{exp} and, consequently, directly affects t_{cpn}. Unlike $\min n$ calculations in the pre-injection–analysis step, efficiency, in that case, influences t_{exp} directly and is *significantly* crucial for each experiment. That means when the method's additional calculation time exceeds the runtime savings, the method is highly inefficient.

Consider the example of predicting the system behavior during the experiment's execution at a predefined time for early failure classification and consequent experiment termination. The method is inefficient if the time taken to calculate predictions exceeds the time saved by early termination. This example also illustrates the need to *separate* effectiveness and efficiency: the method is effective if the prediction of the resulting failure is correct (i.e., the goal of the method), and the method is efficient if the calculation time of this prediction does not exceed the runtime savings from early termination – however, both impact t_{cpn}.

3.2.1.4 Summary of Effectiveness and Efficiency

In summary, the more *effective* an FI-campaign acceleration method is, the more *efficiently* the FI framework produces the desired results for evaluating the SUT's reliability.

The primary focus of this dissertation is to minimize t_{cpn}; effective FI-campaign acceleration methods make the FI-campaign execution more efficient by optimizing $\min n$ and $\min t_{\text{exp}}$.

Throughout this dissertation, I will introduce and discuss three FI-campaign acceleration methods, primarily optimized for their effectiveness. The efficiency of each implemented method is considered and evaluated in different ways, as will be shown in detail in the respective Chapters 4, 5, and 6).

3.2.2 Estimating the Ground Truth with FAIL*

As FAIL* serves as the FI framework used in this work, which generates the desired SUT's reliability evaluation data, FAIL* determines all the parts of the ground truth (refer to Section 3.2.1.1) essential for evaluating FI-campaign acceleration methods.

Figure 3.3 illustrates estimating the ground truth using FAIL*. Consider the code snippet in Listing 3.2 (non-highlighted lines), which produces the 10th Fibonacci number. This exemplary program is subjected to reliability evaluation using FAIL* with the x86 processor-simulator Bochs.

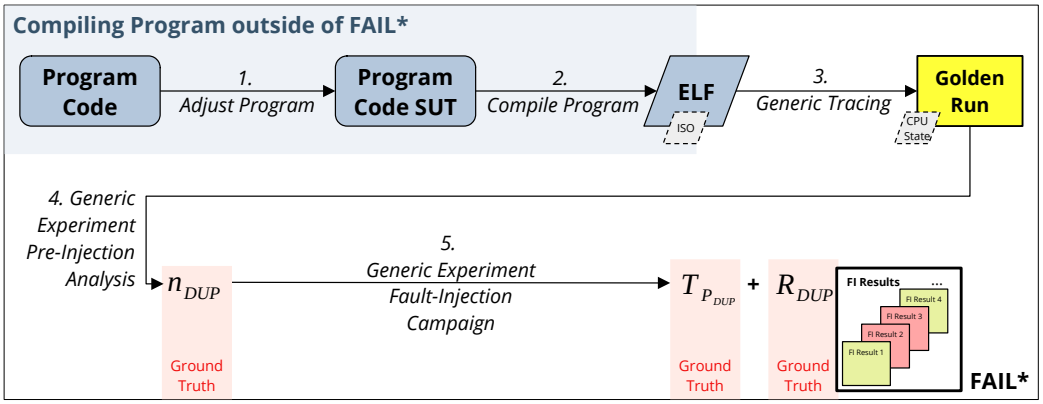


Figure 3.3 – Determining the Ground Truth with FAIL*.
Determining the ground truth with FAIL* encompasses five fundamental steps: (1) The program under evaluation undergoes *adjustments* to ensure compatibility with FAIL*, creating the SUT code. Next, (2) the SUT code is *compiled* into an ELF file and an ISO image. Subsequently, (3) the ELF and ISO files serve as inputs for the FAIL* task *generic tracing*. This step generates the golden run or trace and records the initial state of the CPU. Following the tracing, FAIL* executes its *generic experiment*, comprising a (4) pre-injection analysis that applies DUP to determine the number of pilots (n_{DUP}) for the FI campaign. Finally, (5) FAIL* proceeds to execute the FI campaign. Throughout this step, FAIL* records the individual runtimes of FI experiments ($T_{P_{\text{DUP}}}$) and the resulting outcomes (R_{DUP}). Thus, the determined parts of the ground truth are n_{DUP}, $T_{P_{\text{DUP}}}$, and R_{DUP}.

Adjust Program Adjusting the SUT's program code is the initial step for any FI campaign using FAIL*.[28] Modifications to the code are necessary to enable compatibility with FAIL*, starting with any program, such as the one demonstrated in Listing 3.2.

[28] Most of the simulation-based FI frameworks need adjusted program code. However, these modifications do not impact the pure semantics of the Fibonacci number calculation.

Inserting *dummy functions*, as highlighted in lines 1 to 20, creates explicit *Executable and Linkable Format (ELF) symbols* in the resulting ELF file. FAIL* uses these ELF symbols to control its execution flow, and the symbols serve two primary purposes:

1. For the generic tracing, the tracing scope is defined using the functions `start_trace` (line 33) and `stop_trace` (line 37). These ELF symbols have no semantic significance within the code execution and are solely for tracing purposes without being considered in the trace.
2. Another purpose is to determine the system-failure class (refer to Section 3.1.2) at the end of a *generic experiment*. For this, an if statement, as shown in lines 39 to 45, adjusts the control flow.[29]

These ELF symbols are crucial in subsequent steps for determining the ground truth and do not alter the pure semantics of the program under evaluation.

Listing 3.2 – Exemplary Fibonacci SUT Program Code for FAIL*.
This C++ code computes the n-th Fibonacci number f_n recursively. In this specific example, it calculates the 10th Fibonacci number ($f_{10} = 55$), storing the result in the variable `result`. This final assignment is the output of the specific code segment. The highlighted code is essential for integrating with FAIL*. This extra code adds ELF symbols in the binary file during compilation, making the control flow more controllable. `start_trace` and `stop_trace` define the tracing scope of the function `fib(10)` with the assignment to `result`. `ok_marker` and `fail_marker` are representative failure classifications. The if statement then verifies the result of `fib(10)`. If the result matches `55`, the control flow proceeds to `ok_marker` (failure class *benign*); otherwise, to `fail_marker` (failure class *SDC*). The functions are explicitly defined with `__dummy = 1` to ensure these symbols persist despite compiler optimizations.

```
1  volatile int __dummy;
2  void __attribute__ ((noinline)) ok_marker();
3  void __attribute__ ((noinline)) ok_marker() {
4      __dummy = 1;
5  }
6
7  void __attribute__ ((noinline)) fail_marker();
8  void __attribute__ ((noinline)) fail_marker() {
9      __dummy = 1;
10 }
11
12 void __attribute__ ((noinline)) stop_trace();
13 void __attribute__ ((noinline)) stop_trace() {
14     __dummy = 1;
15 }
16
17 void __attribute__ ((noinline)) start_trace();
18 void __attribute__ ((noinline)) start_trace() {
19     __dummy = 1;
20 }
21
22 unsigned int fib(unsigned int num) {
23     if (num == 0) {
24         return 0;
25     }
26     if (num == 1) {
27         return 1;
28     }
29     return fib(num - 1) + fib(num - 2);
30 }
31 void main() {
32     unsigned int result;
33     start_trace();
34     // trace each instruction behind this line
35     result = fib(10);
36     // stop tracing before this line
37     stop_trace();
38     // compare result of the function fibonacci(10)
39     if (result != 55) {
40         // in case of an incorrect/unexpected result
41         fail_marker();
42     } else {
43         // in case of the correct/expected result 55
44         ok_marker();
45     }
46 }
```

[29] The added code changes the compiled ELF file for technical reasons only, making it traceable and injectable with FAIL*. Listing 3.2 additionally shows the potential level of *intrusion* discussed in Section 2.2.3.1 on page 38.

Compile Program After preparing the SUT's code, this step involves the conversion of the created SUT code into binary files using the *GCC* compiler, resulting in an ELF file and an *Optical Disc Image (ISO)* file. The execution of the SUT code in FAIL* is *bare metal*, meaning that it runs without an operating system. The ELF file – containing the SUT code to execute – needs to be made bootable for the x86 emulator Bochs; the compilation process generates an ISO file to enable Bochs to boot and run the resulting ELF on the emulated x86 processor.

FAIL*'s Generic Tracing This predefined FAIL* task requires an ELF and the corresponding ISO files as input. Using these files, FAIL* generates an output – the *golden run* or *program trace* – derived from code like those in Listing 3.2.

While adjusting the code, specific positions were designated as the trace scope for this FAIL* task using ELF symbols such as `start_trace` and `stop_trace`. FAIL* records each executed ISA instruction after reaching the `start_trace` symbol until it reaches `stop_trace`. FAIL* captures each instruction's IP and memory event, including the corresponding memory addresses.

FAIL* can perform a *full trace* in addition to the regular trace, which also records the values of general-purpose registers and their memory dereferences. FAIL* saves these recorded instructions into a packed file. Listing 2.1 on page 36 illustrates an example of such a recorded golden run.

Additionally, FAIL* records the initial CPU state at the start of execution. This recorded state is essential for the FI campaign as each FAIL* client's emulation (i.e., an FI experiment) begins in this initial CPU state.

FAIL*'s Generic Experiment[30] The core of an FI campaign with FAIL* is importing the previously generated trace into the FAIL* database. This process captures the sequence of IPs, general-purpose–register values, memory information, and access details, which FAIL* stores in its database. Additionally the import step saves an *objdump* from the ELF file, puts it into the database for better readability and maps it to the imported data. This mapping associates an ISA instruction with each imported IP. With the complete trace imported, FAIL* is ready for pre-injection analyses using all potential time points and available locations within the SUT.

Pre-Injection Analysis The pre-injection analysis step analyses the imported information and produces the pilots for the FI campaign. The default method in FAIL* is the DUP. The DUP method calculates and stores its resulting pilots P_{DUP} in the database by processing the trace data. The count of entries in the pilot database table represents the number of pilots n_{DUP} using DUP, which determines the part of the ground truth (i.e., n_{DUP}) for evaluating pre-injection acceleration methods, as described in Equation 3.1.

Fault-Injection Campaign The FI campaign execution is the conclusive step. Once the set of pilots P_{DUP} for the FI campaign is determined, FAIL* executes the campaign.

Initially in this step, FAIL* launches the *job server* to manage the pilot executions.

Subsequently, FAIL* *clients* start their operation. Each client handles the Bochs emulation, the ELF file with the SUT code (along with the requisite ISO for booting), and the initial CPU state. The client initiates the experiment by sending a request to the job server. The job server transmits a job $i \in \{0, \ldots, n_{\text{DUP}} - 1\}$ to the client, containing a pilot $p_i = (t_i, l_i)$ from the pilot database table.

[30] As a reminder of the terms used: The term FAIL* experiment means a complete FI campaign consisting of all individual FI experiments.

The client launches the emulation via Bochs, loads the ELF file, and sets a *breakpoint* at the pilot's designated time point t_i. Until the emulation reaches the breakpoint, it pauses execution, flips the bit at location l_i (refer to Listing 3.1), and resumes executing the SUT. After that, the client reports the observed system behavior to the job server (i.e., the resulting failure classification; refer to Section 3.1.2).

After that, the experiment concludes, and the client requests the job server for a new job. The FI campaign is complete if the job server has no jobs left. The client-server structure allows simultaneous initiation of multiple clients that request jobs from the server and terminate once all jobs are done. The FI campaign duration spans from the start of the job server and the creation of FAIL* clients to the conclusion of the very last FI experiment from the job server.

Each client reports the first and last executed dynamic instruction as well as the start and finish of the experiment . However, this determines the individual pure program duration t_{prg} and the experiment runtime t_{exp}. Without any additional acceleration methods (and additional overhead), all determined experiment durations represent $T_{P_{\text{DUP}}}$ of the ground truth.

After each FI experiment, the SUT's behavior leads to a distinct failure category $c \in C$ of the predefined failure classes under observation detailed in Section 3.1.2. When all FAIL* clients terminate after the job server allocates the last job, the FAIL* database creates a *result table*. This table contains details regarding the outcome of every individual FI experiment, representing the last part of the ground truth R_{DUP}.

3.2.3 Experimental Setup

Following the description of the FI-campaign process and the determination of the ground truth with FAIL*, I describe the *benchmarks portfolio* and the *technical setup* used in this dissertation.

3.2.3.1 Benchmark Portfolio

Benchmarks especially sensitive to bit manipulation are advantageous in effectively evaluating enhancements in FI-campaign runtimes. Bit-sensitive benchmarks are characterized by instruction semantics directly impacting subsequent bit-wise operations computations. Additionally, these instructions possess the potential to mask errors, making the resulting bit flip *benign*. The benchmark portfolio in this dissertation is the following:

MiBench Benchmark Suite The MiBench benchmark suite is a comprehensive collection of embedded benchmarks known for its alignment with commercial applications [Gut+01]. This suite contains diverse categories of benchmarks, including automotive, format encoding, document-specific, networking, security, and telecommunication algorithms.

From the MiBench suite, I selected seven benchmarks from the automotive and security branch. I chose the automotive branch since FIs are recommended to be performed for safety analyses and certifications in the automotive sector [ISO18]. For this branch, I omitted the benchmarks the *masicmath*, since FAIL* does not yet support FIs into floating-point registers, and the benchmark *susan* (image processing), since failure behavior of image recognition is relatively gradual, due to the amount of redundancy within the image than classifiable with a few failure classes. From the security branch, I choose a bundle of different encoding and hash algorithms; these benchmarks are interesting because every single bit is vital for the correct execution

of the algorithm and is thus challenging for FI. I omitted the *pgp* benchmark of this branch because it has properties similar to those of other security benchmarks.

I adjusted the input sizes of these benchmarks to execute about 50000 instructions, achieving a feasible campaign size while ensuring substantial potential for FI-campaign acceleration methods.

The seven selected MiBench benchmarks are as follows:

MiBench Bitcount (miBC) Bitcount assesses a processor's bit manipulation capabilities by computing the bitcount in an array of integers. It consists of five distinct methods to estimate the bitcount. The input data is an array of integers with an equal number of *ones* and *zeros*.

MiBench Blowfish (miBFD/miBFE) Blowfish is a symmetric block cipher featuring a variable-length key. Initially developed in 1993 by Bruce Schneier, its key length ranges from 32 to 448 bits, making it suitable for domestic and exportable encryption. For this benchmark, the input data set is an ASCII text file. The Blowfish benchmark is divided into two separate ones: the decoding algorithm miBFD, and the encoding algorithm miBFE.

MiBench Quicksort (miQS) This quick sort implementation sorts an array of strings in ascending order. Sorting data is crucial for system organization, prioritization, and optimizing program runtimes [Gut+01]. The input for this benchmark is a list of words.

MiBench Rijndael (miRDD/miRDE) The Rijndael encryption is also known as the National Institute of Standards and Technologies Advanced Encryption Standard (commonly AES). It is a block cipher with 128-, 192-, and 256-bit key and block options. The Rijndael benchmark includes decoding (miRDD) and encoding (miRDE) algorithms. The input data set for this benchmarks, similar to miBFD and miBFE, is an ASCII text file.

MiBench SHA1 Checksum (miSHA) SHA1 generates a 160-bit message digest for a given input, commonly used to exchange cryptographic keys and create signatures securely. Like miBFD or miBFE, the input data set for SHA is an ASCII text file.

Micro Benchmarks This collection of benchmarks is self-implemented with algorithms to cover specific algorithm characteristics:

Micro Recursive Fibonacci (μFIB) This benchmark performs the Fibonacci sequence calculation of the n-th Fibonacci number: $f_n = f_{n-1} + f_{n-2}$ with $f_0 = 0$ and $f_1 = 1$ recursively. This implementation primarily consists of memory-indirect control-flow instructions. The input required for this algorithm is an integer n.

Micro Iterative Looped Sum (μLSUM) This benchmark calculates an iterative summation of an array of integers. It obtains each item i_j in this sequence through an arithmetic calculation with a constant c and a value v_j retrieved from an array: $i_j = v_j \cdot c + 1$. Notably, this benchmark produces almost no intermediate results. Its input is an array of integers.

Micro Mixed Bit-Wise Operations (μMIX) μLSUM is the basis for this benchmark, and μMIX consists of both the calculation and the array referenced of the μLSUM benchmark. In addition to these elements, μMIX contains numerous bit-wise logical operations, the outcomes of which are susceptible to individual bit flips. These operations are `ADD`, `AND`, `OR`, and `XOR` instructions; the benchmark applies them to items i_j. Like μLSUM, the input data for μMIX is an array of integers.

Micro Quicksort (μQSI/μQSR)[31] These quick sorts compute the distances of n three-dimensional vectors of integers from the coordinate origin and then sort these vectors in ascending order based on these calculated distances. The 2-norm[32] determine the distance d of a vector $\vec{x}_i = (x_{i,0}, x_{i,1}, x_{i,2})$ among the n vectors. equation $d(\vec{x}_i) = \sqrt{x_{i,0}^2 + x_{i,1}^2 + x_{i,2}^2}$. The quick sort algorithm exist in two variants: Iteratively (μQSI) and recursively (μQSR). The input data for this benchmark is an array containing $3n$ integers, which represent n three-dimensional vectors.

Table 3.1 shows the central characteristic and specific attributes of the mentioned benchmarks, which include details such as the count of instructions in the golden run ($|T|-1$), the size of the FS denoted as $|\mathscr{F}| = |T \times L|$, the number of DUP pilots denoted as n_{DUP},[33] as well as the *effective* FS $\mathscr{E}$ relating to the application of the DUP.

Benchmark	Central Characteristic	Fault Space			DUP
		Instructions	$\lvert\mathscr{F}\rvert$ $[10^6]$	$\lvert\mathscr{E}\rvert$ $[10^6]$	n_{DUP} $[10^3]$
miBC	mostly bit-wise operators	54 434	80.04	70.33	2 521.85
miBFD	mostly bit-wise operators	55 647	2 173.23	1 894.80	3 501.60
miBFE	mostly bit-wise operators	54 951	2 155.18	1 880.81	3 457.00
miQS	data-controlled CFs	45 774	1 776.21	1 623.90	2 931.66
miRDD	mostly bit-wise operators	70 824	3 851.64	3 506.16	4 172.74
miRDE	mostly bit-wise operators	70 450	3 808.26	3 457.58	4 169.90
miSHA	mostly bit-wise operators	40 647	342.43	242.63	2 672.68
μFIB	memory-indirect CFs	2 093	2.33	1.13	100.49
μLSUM	sparse intermediate results	54	0.02	0.02	3.50
μMIX	mostly bit-wise instructions	90	0.05	0.03	5.45
μQSI	data-controlled CFs	800	1.59	1.20	46.79
μQSR	data-controlled CFs	804	1.35	0.91	46.70

Table 3.1 – Overview of the Benchmark Portfolio Characteristics.
The table summarizes attributes of the benchmarks outlined in Section 3.2.3.1 with their central characteristic (CF stands for control flow). Each benchmark's entry includes the instructions count, corresponding to the number of time points $|T|-1$ in the whole FS $\mathscr{F} = T \times L = n_{\text{naive}}$. The subset of the FS $\mathscr{E}$ corresponds to the part of the FS that is *effective*. Lastly, the column on the very right shows the count of pilots n_{DUP} after applying the DUP acceleration method.

3.2.3.2 Technical Setup

FAIL* can be used with various technical setups and hardware configurations. Given the potentially high computational demands of an FI campaign (refer to Section 2.3.1 on page 45 for an understanding of the infeasible upper bound), it is advantageous to use a robust computing system or even a cluster, particularly for step C in Figure 3.1 for both the determination of the ground truth and the calculation of SUT reliability evaluation data.

[31] These Micro benchmarks are inspired by a MiBench quick sort variant.

[32] In the context of the 2-norm calculation, the *integer* square root uses a modified version of Newton's method for approximating roots for discrete ranges of numbers since FAIL* yet does not support FIs in float numbers.

[33] The table in Table 3.1 shows the substantial reduction in the number of pilots by two orders of magnitude [Bar+05] as discussed in Section 2.3.2.1. This reduction is evident when comparing the result obtained from the naive approach, $n_{\text{naive}} = |\mathscr{F}|$, which is significantly larger than the count after applying the DUP method, n_{DUP}.

The client-server structure of FAIL*, combined with the data independence of the FIs, allows for a highly parallelized execution of FI campaigns through FAIL*. For this work, FAIL* executed the required FIs on several systems. For instance, the clients operated on a cluster equipped with 17 Intel X5650 @ 2.67 GHz (12 cores each), enabling the simultaneous execution of 204 FIs, and additionally on systems with Intel Xeon Gold 5320 @ 2.2 GHz (26 cores) or Intel Xeon Gold 6252 @ 2.1 GHz (24 cores).

Moreover, configuration, FAIL*'s compilation, pre-injection, and post-injection analyses are less computationally intensive. For these operations, FAIL* executed these computations single-threaded on an Intel i5-7400 @ 3 GHz (4 cores) or on an Intel i5-6400 @ 2.7 GHz (4 cores).

3.3 Summary of Implementation and Evaluation Process

This chapter described the crucial aspects for comprehending my contributions and their evaluation.

Initially, I described the single steps of the FI framework FAIL*, a C++ implementation of the FI-campaign process (refer to Section 2.2.2 on page 35). The essential steps for this work are the *pre-injection analysis* (step B in Figure 3.1) and the execution of the campaign's *FI experiments* (step C in Figure 3.1). These steps are technically extendable using *FAIL* plugins* and *FAIL* tools*, aiming to make FI campaigns more efficient. I integrated my *contributions* as FAIL* plugins and FAIL* tools, allowing for assessing their effectiveness and efficiency. Additionally, I presented the *failure classes* considered in the context of this work using FAIL*.

I outlined the meaning of *effectiveness* and *efficiency* in the context of FI-campaign acceleration methods, a comprehension that needs a solid foundation – the *ground truth*. Following the conceptual exploration of effectiveness, efficiency, and ground truth, I delved into determining this work's ground truth using FAIL*. The ground truth encompassed three parts in the pre-injection step: (1) the determination of DUP pilots or rather their quantity n_{DUP}; during the execution of the resulting individual FI experiments, (2) the recording of all pure experiment runtimes $t_{\mathrm{exp},i}$; and after the completion of the campaign, (3) the set of all classified system failures post-FI R_{DUP}. These three parts are indispensable for assessing the effectiveness of my contributions.

Additionally, I provided insights into this work's overall evaluation setup, outlining the selected benchmarks from MiBench and some self-constructed Micro benchmarks and describing the technical setup for the evaluations.

4

Data-Flow–Sensitive Fault-Space Pruning

> Give me something to assemble, I won't look at the directions, I'll try to figure it out by myself. It's why I love Ikea furniture.
>
> DAVID ERIC "DAVE" GROHL (BORN 1969)

This chapter delves into the primary extension of the FI framework FAIL* within the context of this dissertation: the *Data-Flow–Sensitive Fault-Space Pruning*. Unlike the DUP, the *Data-Flow Pruning (DFP)* takes a more comprehensive approach by considering not only read and write accesses in the golden run but also incorporating the program's *data flow*. This combination involves exploring the *propagation* and *masking effects* on instructions and analyzing the data- flow graph and its instructions.

The inspiration behind the Data-Flow–Sensitive Fault Space Pruning method stems from my prior collaboration on the publication by Dietrich et al. [▷Die+18]. This research explored masking effects in a cross-layered context, devising efficient injection optimizations. In alignment with FAIL*'s focus on the ISA layer, DFP not only considers masking effects but also investigates the potential *propagation* of a bit flip from an input value to an output value of an ISA instruction, leading to the identification of new extended equivalence sets. These individual propagations can be examined across the data flow, facilitating a comprehensive assessment of propagation and masking effects throughout the entire program's data flow.

By *globally* analyzing these effects, we gain deeper insights into how bit flips can propagate and be masked within the data flow, ultimately resulting in more efficient fault space pruning. To accomplish this, I developed the DFP method and will detail it in this chapter.

The research thoroughly examined propagation and masking effects for a select set of instructions. Insights from this analysis were integrated into a developed DFP process and implemented as a tool within the FAIL* framework. Additionally, the corresponding evaluation results allow one to assess the *effectiveness* of DFP, demonstrating significant advantages such as a notable reduction in the FI campaign duration with minimal overhead.

DFP combines the strengths of the DUP's fault equivalence sets, making it an *enhanced* version of the DUP.

I presented the DFP at the LCTES conference in 2021 [▷PDL21] and am the paper's primary author. I conceptualized and implemented the DFP, designed benchmarks, conducted experiments, and contributed to the paper's writing.

4.1 Data-Flow–Aware Fault-Equivalence Set

The widely used DUP (see Section 2.3.2.1 on page 47) combines potential FIs from the FS into FESs, with only one representative *pilot* of each FES for full FS coverage. However, DUP uses only the information from read and write accesses of individual instructions in the trace, which results in one-dimensional FESs (temporal dimension). As a result, DUP cannot form two-dimensional FESs (temporal *and* location dimensions) that span over various locations without the data-flow information. Whereas the time and location of each value read or written by an instruction are known, the information about the data flow, including the semantics of the instructions and the concrete values used, is missing. Both factors have a decisive influence on the propagation of faults in the system. Therefore, it is helpful to consider these factors when adjusting the FESs, rather than relying solely on read and write accesses of the instructions, as in the well-known DUP.

4.1.1 Instruction Awareness

Suppose we are dealing with a system having two 4-bit registers, and the program executed consists only of the `MOV` instruction. The `MOV` instruction transfers data from one register to another, from `R1` to `R0`. The data in `R0` is the program's output.

Figure 4.1a illustrates this system's FS when DUP is applied. The system is injectable both before and after the `MOV` instruction in all eight bits of both registers. This results in an FS of size $|\mathscr{F}| = 2 \cdot 8 = 16$, corresponding to 16 potential FIs.

The `MOV` instruction reads data from `R1`, which is then available in `R0` as the system output. Applying DUP allows us to determine the effective FS and its corresponding FESs and weights, resulting in eight FESs for this example. As a result, the number of required FIs is 8.

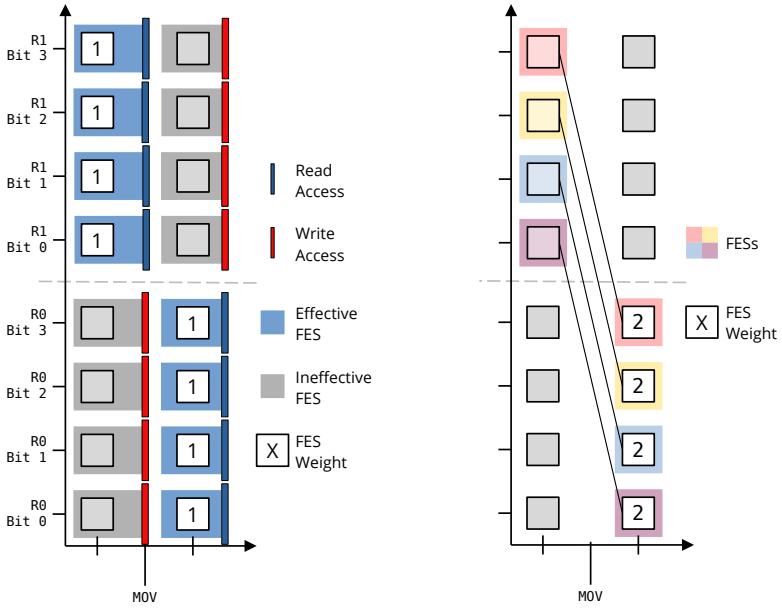
However, this reduction is only one-dimensional. By knowing the semantics of the `MOV` instruction, which copies data from one location to another unchanged, we can unify the FESs obtained through DUP. Figure 4.1b visualizes this unification, including the register bit values. For instance, a bit flip in bit 0 of register `R1` is equivalent to a bit flip in bit 0 of register `R0` after executing `MOV`. As a result, the number of FESs and the corresponding FIs required for this FS is reduced to now 4. The resulting FESs in this example now consider multiple locations in the FS. and thus become two-dimensional.

4.1.2 Value Sensitivity

Another essential aspect that needs to be considered in the DUP is the effect of concrete data values. The examples in Figure 4.2 illustrate how to consider data values when executing a 1-bit `OR` instruction. The figure shows all possible inputs and the corresponding output of the `OR` instruction. This consideration determines fault equivalences in the input and output, depending on the input data values and the `OR` instruction semantics.

For example, in the first case shown in Figure 4.2, if any bit in the input flips, the output value changes from *zero* to *one*, the same result as if the output bit flips. Therefore, both *zero* inputs and the output are equivalent in the occurrence of bit flips. In the second example, a flip in the *one* in the input directly influences the output, so the input with the *one* and the output is considered equivalent regarding bit flips. On the other hand, a flip in the *zero* in the input in the second example does not affect on the output and is considered *ineffective* or *benign*. The other two examples follow a similar pattern.

By taking data values and the semantics of the instruction into account, it is possible to create a mapping for *instruction-local* FES between inputs and outputs of the instruction.



(a) Def/Use Pruning **(b)** Instruction-Aware FESs

Figure 4.1 – Instruction-Aware Fault-Equivalence Sets.
In Figure 4.1a, after applying the DUP technique, we see the FS of a system with two 4-bit registers executing a `MOV` instruction that copies data from `R1` to `R0`. This results in 8 effective FESs with weight 1 and thus 8 FIs. Figure 4.1b shows the same FS but with the semantics of the `MOV` instruction and the concrete bit values considered. As a result, only 4 effective FESs with weight 2 are determined; thus, the number of necessary FIs is reduced to 4.

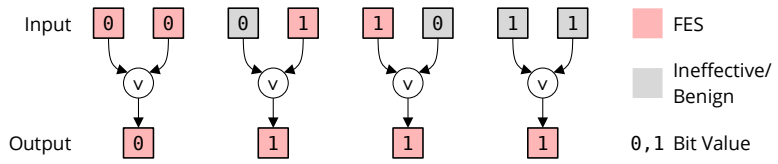
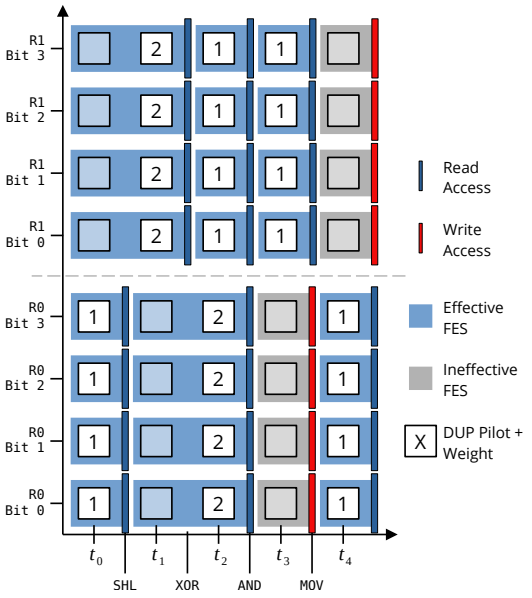


Figure 4.2 – Value-Aware Failure Equivalence Sets.
The figure shows all possible input variations and their corresponding output bits for a bit-wise `OR` instruction. If a bit in the input or output flips, certain input bits or the output bit are considered equivalent to an occurring bit flip. In the first example, flipping a bit in the input results in a *one* in the output, as if the output bit flips directly. Therefore, in the first example, all existing bits are combined into one FES. In the second example, the output bit is not affected by flipping the first input bit due to the semantics of the `OR` instruction; this bit flip is *benign*.

4.1.3 Exemplary Data-Flow–Aware Fault-Equivalence Sets

To demonstrate the benefits of incorporating *instruction awareness* and *value sensitivity* into DUP, I present the example code from Listing 4.3b, which includes four instructions and results in five distinct system states, t_0 to t_4. The system contains two 4-bit registers again. Applying DUP generates the FS shown in Figure 4.3a, which contains the resulting ineffective FS $\mathscr{I}$ and the resulting FESs. The original FS size is $|\mathscr{F}| = 5 \cdot 8 = 40$ possible FIs, and only 24 FIs are required after applying DUP.



(a) Def/Use Pruning

```
1   # t_0: R0=5, R1=11
2
3   # shift-left by 1
4   R0 := SHL R0, 1
5
6   # t_1: R0=10, R1=11
7
8   # bit-wise XOR with 7
9   R1 := XOR R1, 7
10
11  # t_2: R0=10, R1=12
12
13  # bit-wise AND
14  R1 := AND R0, R1
15
16  # t_3: R0=10, R1=8
17
18  # move result to R0
19  R0 := MOV R1
20
21  # t_4: R0=8, R1=8
22
23  # result in R0
```

(b) Example Code

Figure 4.3 – Application of Def/Use Pruning to Example Code with Multiple Instructions. Figure 4.3a shows the FS and the result of applying DUP with the executable code presented in Listing 4.3b. The comments in the code show the values of the registers between each instruction.

As outlined in the previous sections (4.1.1 and 4.1.2), incorporating instruction semantics and data values can enhance the construction of FESs. The example pseudo-code in Listing 4.3b includes the instructions `SHL` (shift left), `XOR` (bit-wise exclusive-OR), `AND` (bit-wise AND), and `MOV` (move data). Instructions such as `SHL` and `MOV`, which move bits between locations, can unify FIs by leveraging knowledge of instruction semantics across locations, as explained in Section 4.1.1. Information about the data values is unnecessary for such instructions, as their semantics are *value-insensitive*. The `XOR` instruction is another example of a value-insensitive instruction, as its semantics allow all input bits, as well as the output bit, to be combined into a single FES regardless of the input bit values because any change in the input bits, regardless of their value, has a direct effect on the output bits. For instructions like `AND`, knowledge of the used data, as described in section 4.1.2, is necessary in addition to the instruction semantics, which is indeed value-sensitive.

By knowing the individual additional *instruction-local* FES constructing rules, it becomes feasible to combine them across the instructions of the entire program. Figure 4.4 demonstrates such combinations, which exhibit the FS depicted in Figure 4.3a, along with the data bits of the registers and the enhanced FESs across all instructions; the edges between the bits represent the unified bits of the enhances FESs. The construction of the FESs for each program instruction in Listing 4.3b for the FESs is as follows (I will revisit the topic of instruction-local FESs in detail later in Section 4.2.2):

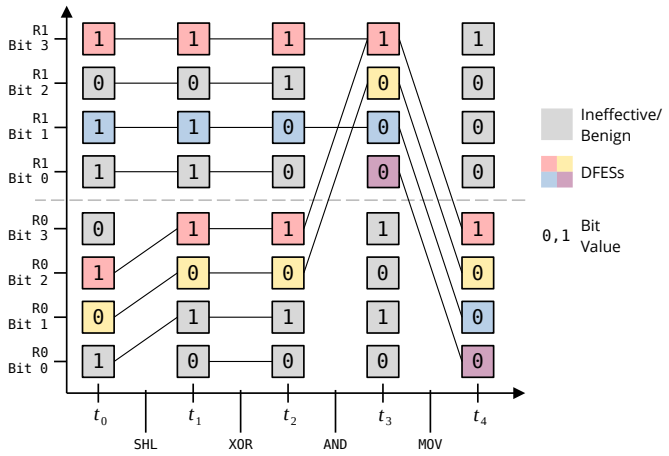


Figure 4.4 – Visualizing Fault Space with Data-Flow–Aware Fault-Equivalence Sets.
This figure illustrates the same FS as in Figure 4.3a. It shows the values of each bit at all times and locations of the two registers. The connections between the bits represent the instruction-local FESs, which indicate the propagation flow of a bit flip. All points in the FS that are reachable from the output (at time t_4 and register `R0`) by the instruction-local FESs are combined into higher-level DFESs. The remaining points in the FS belong to the ineffective FS $\mathscr{I}$, as compared to Figure 4.3a, which applied the DUP technique, the number of required FIs is significantly reduced from 24 to 4.

`r0 := SHL r0, 1` shifts the bits within the data word of the register `R0` to the left by one position. For instance, it shifts bit 0 to bit position 1 after executing the `SHL` instruction. Therefore, any bit flip at time t_0 in bit 0 will appear at time t_1 in bit 1. Similar to the `MOV` instruction, it is possible to construct FESs across two different locations. In this case, the FES includes both bit 0 at time t_0 and bit 1 at t_1.

`r1 := XOR r1, 7` performs an immediate exclusive-or instruction with the value $7_{10} = 0111_2$ on the data stored in register `R1`. This results in the computation $1011_2 \oplus 0111_2 = 1100_2$ (on the left side of the $\oplus$ is the input value of `R1`). The `XOR` instruction immediately reflects any alteration at its input, whether caused by switching the input value or by a bit flip due to transient effects. Therefore, it is possible to combine each bit of the input register `R1` with the corresponding bit of the output to form an FES. If the second input of the `XOR` instruction were another register instead of an immediate value, each bit of the second register could also be combined bit-wise with the corresponding FESs.

`r1 := AND r0, r1` is analogous to the `OR` instruction presented in Section 4.1.2, and the corresponding FESs are constructed similarly. Combining the respective bits at position 3 of the input and output is possible since any changes in the bits of the equation $1 \wedge 1 = 1$ lead to the outcome *zero*. The masking effects in the input bit combinations $1 \wedge 0$ or $0 \wedge 1$ can also be mapped into FESs similarly to the `OR` case from Section 4.1.2. Moreover, for the input bit combination $0 \wedge 0$, a bit flip at the inputs is considered *benign*, as the output bit will not change. Planning for injections at these bits $0 \wedge 0$ is still necessary when applying DUP. However, it is unnecessary since bit flips can be classified as inside the ineffective FS $\mathscr{I}$ by the instruction semantics and the input values at the point.

r0 := MOV r1 represents the same pattern as already described in Section 4.1.1. Since this instruction is value-insensitive and only moves data, constructing location-crossing FESs is relatively straightforward.

The rules that construct instruction-local FESs link propagations of bit flips across all program instructions up to the output at time t_4. When considering all instruction semantics, input and output values of the instructions, and concatenations with each other globally over the program, this corresponds effectively to considering the *data flow* of a program. With knowledge of the data flow, one can trace the *global propagation* of bit flips and group them into *Data-Flow–Aware Fault-Equivalence Sets (DFESs)*. In the example of Listing 4.3b, we see the DFESs shown in Figure 4.4 using the abovementioned scheme. Each bit, which could propagate a bit flip to the output, belongs to a DFES, considering instruction semantics or value sensitivity. Therefore, all other points in the FS are deemed *ineffective* and do not require consideration by FIs. In the case of Listing 4.3b, the number of FIs required to cover the effective FS entirely drops significantly from 24 to just 4.

This brief introduction provides insight into the *Data-Flow Pruning (DFP)* [▷PDL21], which is one of the contributions of this work. DFP follows the deterministic rules of the instruction semantics, so it is a precise technique that completely covers the FS.

4.2 Data-Flow–Sensitive Fault-Space Pruning Process

The Data-Flow–Sensitive Fault-Space Pruning is a precise technique that prunes the number of necessary FIs to cover the FS by using the program's *data flow* and the instructional semantics with concrete values, as presented in Section 4.1. Since DFP follows the deterministic semantics of the instructions, the resulting DFESs are also determined deterministically. The DFP involves several steps, which are detailed in this section and have been implemented as a FAIL* tool. This section aims to gain a more profound comprehension of the DFP technique's functionality and explore possibilities for further optimization by thoroughly examining the data flow to construct DFESs.

4.2.1 Data-Flow Graph

During program execution, instructions process data, with each instruction requiring input data and producing output data that depends on the instruction's semantics. Instructions manipulate or transfer this data as it flows through the program, such as through arithmetic or logical computation or data movement via the instructions.

A directed graph called the *Data-Flow Graph (DFG)* represents the program's data flow, with nodes representing data values and instructions and directed edges indicating the direction of data movement. Figure 4.5 shows the DFG for the program from Listing 4.3b, where input data nodes connect to instruction nodes, and all directed edges show the flow of data through the DFG. Some instructions have a fixed immediate value hard-coded to the instruction by an opcode, like the `SHL` instruction in Listing 4.3b. Once the program finishes executing, a data node at the output, here at t_4 register `R0`, represents the program's output. The horizontal length of the data nodes in the DFG represents the lifetime of the data (i.e., how long the data remains in the program without being overwritten by instructions).

4.2.2 Instruction-Local Fault-Equivalence Set

Each instruction in the DFG has the potential to propagate bit flips from its input to its output. Instruction-specific *mappings* capture this behavior that generates instruction-specific FESs, intro-

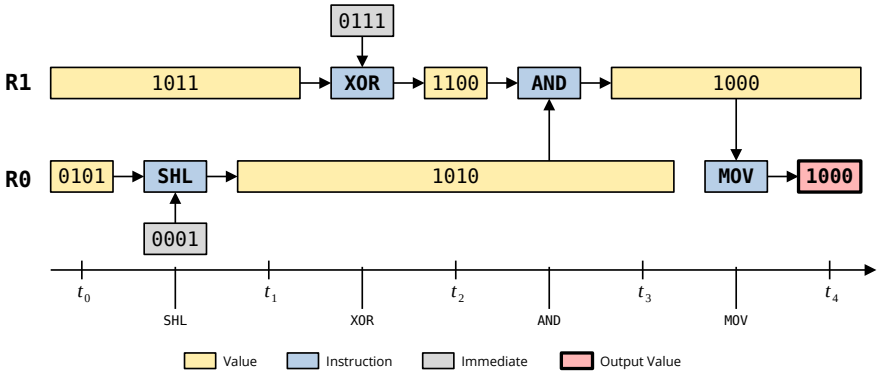


Figure 4.5 – Data-Flow Graph.
This figure illustrates the data-flow graph for the program from Listing 4.3b, consisting of data and instruction nodes and directed edges representing the data flow. Data nodes serve as input for instruction nodes, which point to other data nodes. Some instructions, such as `SHL`, have a hard-coded immediate value. The program's output is represented by a data node, in this case the data node at register `R0` in time step t_4. The horizontal length of the data nodes corresponds to the time in the program during which instructions have not overwritten the data.

duced as *Instruction-Local Fault-Equivalence Sets (IFESs)* in this thesis, based on the instruction semantics and input data. In this subsection, I will first introduce the mapping notation used to construct IFESs in general. Then, I will present specific mappings developed for five different instructions used in the implemented FAIL* tool DFP.

4.2.2.1 Instruction-Local Fault-Equivalence Set Mapping

With knowledge of the instruction and its semantics, a suitable instruction and input configuration *mapping* generates an IFES. The general format of such a mapping is as follows:

$$i_1 \square i_2 \rightarrow o \Rightarrow s_1 \circ s_2 \rightarrow s_3$$

An arbitrary instruction represented by the $\square$ with input bits i_1 and i_2 produces the output bit o. Examining all possible input bit permutations is necessary to identify all potential IFESs for this instruction. With two input bits, there are $2^2 = 4$ possible input configurations. Thus, it is necessary to construct four mappings for the instruction $\square$.

After constructing the four mappings for the instruction $\square$, each mapping leads to corresponding IFESs, represented by unique symbols s_i. The $\circ$ acts as a separator between the input symbol, usually non-commutative unless the instruction $\square$ is commutative. The term $s_1 \circ s_2 \rightarrow s_3$ initially represents the creation of three different IFESs. To identify equivalent IFESs, the instruction semantics of $\square$ are analyzed to determine whether bit flips on the input lead to the same output. For instance, the term $0 \wedge 1 \rightarrow 0$ shows that changing the first input bit i_1 is equivalent to changing the output o, whereas changing the second input bit i_2 has no effect and is considered benign. Therefore, in this case, s_1 and s_3 can be equated, resulting in $s_3 \circ s_2 \rightarrow s_3$. Thus, two IFESs occur: The IFES containing i_1 and o symbolized with s_1 and the other IFES named with s_3, which contains only i_2. Since flipping i_1 leads to flipping the output bit o, and o is also in the IFES, I refer to this kind of IFES as *effective*. The other IFESs are *ineffective* or *benign* since they do not contain o since the flipped bit does not propagate to the output by the instruction. Since the input bits indeed can only have two possible values (*zero* or *one*), and each input bit can either be flipped or not flipped, there can only be a maximum of two

possible IFESs per mapping, one for the case where the output bit is flipped (effective) and one for the case where it is not flipped (benign).

If an input to the function $\square$ is an immediate value, the value is used to calculate the output, but the mapping does not equate the corresponding symbols. They are left untouched because, on the ISA layer, immediate values are not subject to FIs, and thus, it is not necessary to investigate how a bit flip propagates on such an input.

4.2.2.2 Instruction-Local Fault-Equivalence Set Mappings in this Thesis

In the context of this thesis, I have selected five instructions to implement in the DFP. These instructions are either the most basic, such as `MOV` and `ADD`, or additionally bit-sensitive logical instructions, such as `XOR`, `AND`, and `OR`. The chosen instructions occur in 26 to 68 percent of the executed instructions within my benchmark portfolio (I will revisit this in Section 4.3.1). This percentage is adequate for the initial assessment of the DFP's effectiveness.

I will present the IFES mappings for these five instructions in the following. In this section, α stands for symbols in a effective IFES and β for symbols in a benign IFES.

MOV

As mentioned in Section 4.1.1, the instruction `MOV` simply copies bits from one location to another. From a bit-wise perspective, `MOV` has only one input that is directly is use. Therefore, a bit flip in the copied bit affects the target register as if a bit in the target register had been flipped directly. Therefore, each bit involved in the instruction belongs to an effective IFES from a bit-wise perspective.

$$b \rightarrow b \Rightarrow \alpha \rightarrow \alpha \mid b \in \{0,1\}$$

Instructions like `SHL` in Listing 4.3b, which shift data, work similarly. In the case of `SHL`, only the bit overwritten by the shift in the register must be ignored since the bit flip does not propagate; Figure 4.4 at t_0 in bit 3 of `R0` illustrates this.

XOR

The `XOR` instruction is a logical operation that takes two input values and produces a single output value based on the following rule: If the inputs differ, the output is *one*; if the inputs are equal, the output is *zero*.

$$0 \oplus 0 \rightarrow 0 \Rightarrow \alpha \circ \alpha \rightarrow \alpha$$
$$0 \oplus 1 \rightarrow 1 \Rightarrow \alpha \circ \alpha \rightarrow \alpha$$
$$1 \oplus 0 \rightarrow 1 \Rightarrow \alpha \circ \alpha \rightarrow \alpha$$
$$1 \oplus 1 \rightarrow 0 \Rightarrow \alpha \circ \alpha \rightarrow \alpha$$

Since each input bit can take only two values, any change in their values will necessarily result in a toggled output. Therefore, any configuration of the input results in both input bits being in the effective IFES with the output bit.

AND

Bit flips in the inputs of the `AND` instruction may have varying effects depending on the input configuration.

$$0 \wedge 0 \rightarrow 0 \Rightarrow \beta \circ \beta \rightarrow \alpha$$
$$0 \wedge 1 \rightarrow 0 \Rightarrow \alpha \circ \beta \rightarrow \alpha$$
$$1 \wedge 0 \rightarrow 0 \Rightarrow \beta \circ \alpha \rightarrow \alpha$$
$$1 \wedge 1 \rightarrow 1 \Rightarrow \alpha \circ \alpha \rightarrow \alpha$$

Flipping the inputs in $0 \wedge 0 \rightarrow 0$ will not affect the output, as both input bits must be *one* to propagate a bit flip to the output. As a result, only the output bit is in an effective IFES, and the symbols are ineffective β. Thus, in this instruction and input configuration, no propagation of bit flips occurs. In configurations $0 \wedge 1 \rightarrow 0$ and $1 \wedge 0 \rightarrow 0$, a bit flip in an input with the value *zero* propagates to the output, causing it to toggle (i.e., bit flips at this input bits are effective). The other output with the value *one* remains unaffected and, therefore, is benign, marked with β. However, in the last input configuration, $1 \wedge 1 \rightarrow 1$, all bits involved belong to an effective IFES, meaning that any bit flip in any input will cause a toggle at the output.

OR

The `OR` instruction is the logical opposite of the `AND` instruction, and the mappings of the previous section are transferable accordingly.

$$0 \vee 0 \rightarrow 0 \Rightarrow \alpha \circ \alpha \rightarrow \alpha$$
$$0 \vee 1 \rightarrow 1 \Rightarrow \beta \circ \alpha \rightarrow \alpha$$
$$1 \vee 0 \rightarrow 1 \Rightarrow \alpha \circ \beta \rightarrow \alpha$$
$$1 \vee 1 \rightarrow 1 \Rightarrow \beta \circ \beta \rightarrow \alpha$$

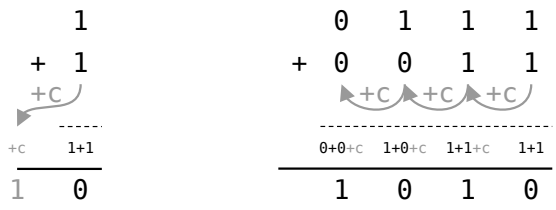
Figure 4.2 visualizes the effective IFES for the `OR` instruction.

ADD

The `ADD` instruction is as essential arithmetic operation that is fundamentally a bit-wise logical `XOR` operation. In the case of a bit-wise addition of $1_2 + 1_2$, the number of bits is insufficient to represent the tow-bit result 10_2. On the circuit layer, half-adders use a *carry* signal set if there is a value *carry over* from the addition. Figure 4.6a illustrates the 1-bit addition with a carry. The resulting bit is *zero*, as it is also output from the `XOR` instruction, and a carry is set to make the arithmetic result 10_2 representable.

However, the typical use case of `ADD` is adding entire data words that represent numbers, addresses, or other values. Figure 4.6b presents an example of adding 4-bit words. Bit-wise addition from the least significant bit to the most significant bit can result in a carry from one bit position to the next; at the circuit layer, the concatenation of half-adders creates full adders. If a carry occurs after the most significant bit position, a flag is set in the arithmetic logical unit to make the result representable.

Carry signals between bits are not directly accessible at the ISA layer and thus cannot be intentionally targeted for FIs at that layer. However, carries between bits exist *virtually* and are still crucial when constructing IFESs. In other words, any bit flip that occurs in a position where a carry occurs during addition can affect the final result of the addition, even if that bit is not directly involved in the addition operation itself.



(a) Bit-wise ADD **(b)** 4-Bit Word-Wise ADD

Figure 4.6 – Visualizing the Carry of Exemplary `ADD` Instruction.
During a bit-wise addition, the addition of $1_2 + 1_2 = 10_2$ results in a carry bit, representing a higher-order bit semantically, as demonstrated in Figure 4.6a. In a word-wise addition like in Figure 4.6b, single carries (+ c) are transferred to the next higher-order bit and used to calculate the corresponding bit value until reaching the most significant bit.

The mapping considers the virtual carries to construct IFESs for the `ADD` instruction, as they hold semantic meaning for the result of a bit-wise addition within a word-wise addition. In the case of the bit-wise addition $1_2 + 1_2 = 10_2$ shown in Figure 4.6a, a carry occurs, which is relevant for the following bit. If one of the inputs flips, the carry is dropped, and the output bit flips analogous to the `XOR` instruction. If the output flips, a *one* is also created but still with the carry. However, the determination of IFESs differs from the `XOR` instruction in that it considers the carries additionally. All bits at inputs and outputs of a bit-wise `ADD` instruction belong to effective IFESs but must be distinguished by the resulting carry when a bit flip occurs. To represent this distinction in the symbols for this instruction, I introduce κ, which indicates that after the bit flip, a carry exists, independent of whether the bit flip caused this carry. Otherwise, it is α as usual. For bits marked with α, a regular carry may be dropped after the bit flip.

When creating IFESs, it is imperative to consider the presence of any carry from the previous bit-wise addition in the calculation. The initial mappings of Equation 4.1 are relevant for the least significant bit.

$$\begin{aligned} 0+0 &\to 0 && \Rightarrow \alpha \circ \alpha \to \alpha \\ 0+1 &\to 1 && \Rightarrow \kappa \circ \alpha \to \alpha \\ 1+0 &\to 1 && \Rightarrow \alpha \circ \kappa \to \alpha \\ 1+1 &\to 0 + \mathrm{c} && \Rightarrow \alpha \circ \alpha \to \kappa \end{aligned} \tag{4.1}$$

If a carry arises from the calculation, regardless of whether it resulted from a bit flip or not, the mappings of Equation 4.2 remain applicable. Notably, the creation of mappings also hinges on the outcome of the preceding calculation, similar to the carry semantic itself. In summary, if a carry results from the calculation and bit flip, or rather κ represents the IFES, the mappings of Equation 4.2 are relevant; otherwise, the mappings from Equation 4.1 are applied.

$$\begin{aligned} \mathrm{c} + 0+0 &\to 1 && \Rightarrow \kappa \circ \kappa \to \alpha \\ \mathrm{c} + 0+1 &\to 0 + \mathrm{c} && \Rightarrow \kappa \circ \alpha \to \kappa \\ \mathrm{c} + 1+0 &\to 0 + \mathrm{c} && \Rightarrow \alpha \circ \kappa \to \kappa \\ \mathrm{c} + 1+1 &\to 1 + \mathrm{c} && \Rightarrow \kappa \circ \kappa \to \kappa \end{aligned} \tag{4.2}$$

4.2.2.3 Generic Algorithm for Determining Instruction-Local Fault-Equivalence Sets

Effective IFESs can be determined *algorithmically*. Listing 4.1 presents a Python-like pseudo-code for determining the effective IFES for any function □.

The objective is to identify all possible IFESs for the bit-wise function implemented in the `square_function` (represents □) starting from line 47; this code implements the `OR` instruction in this exemplary code. The `ifes` function is called in `compute_all_ifes` from line 29, considering all $2^2 = 4$ potential input configurations to determine the complete set of all IFESs. The `ifes` function sets unique values for the symbols s_1, s_2, and s_3. The IFES initially contains the output symbol s_3. The goal is to determine if and which input symbols, s_1 and s_2, belong to the IFES. The function `ifes` calculates the correct result of the `square_function` based on the given input bits i_1 and i_2. When an **`if`** statement in line 16 or 22 is **`True`** and the corresponding flipped input is not an *immediate value*, the algorithm adds the corresponding symbol to the effective IFES `eff`. After `compute_all_ifes` considers all bit configurations, all IFESs are calculated, and a mapping exists, as displayed in line 52.

For instance, the mapping `((0,1), {2, 3})` represents $0 \vee 1 \rightarrow 1 \Rightarrow s_1 \circ s_3 \rightarrow s_3$, which means that s_2 and s_3 (second input and output) are in an effective IFES, and a bit flip at the first input bit is ineffective, thus s_1 is not in the set of the effective IFES. The output from line 52 is equivalent to the equivalences already visualized in Figure 4.2 on page 74.

Listing 4.1 – Determining Instruction-Local Fault-Equivalence Sets for an Arbitrary Function.
This listing shows a Python-like pseudo-code for determining local IFESs. After defining the `square_function` with the operation under consideration, `compute_all_ifes` starts the determination. This function calls the `ifes` function four times, with each possible input bit combination and determines the effective IFES for a given input bit configuration. After considering all input combinations, the code displays a complete IFES mapping. The highlighted parts (`func`, `square_function`) the segments where the code passes or calls the `square_function`.

```
1  def ifes(func: BoolFunction, i1: bool, i2: bool) ->
   ↪ set:
2      # init three different symbols
3      s1 = SYMBOL_I1
4      s2 = SYMBOL_I2
5      s3 = SYMBOL_O
6
7      # init effective IFES
8      eff = set(s3)
9
10     # compute the output bit
11     o = func(i1, i2)
12
13     # check if flipping the first input bit
14     # leads to the same result
15     # as flipping the output bit
16     if func(not i1, i2) == not o
17         and not is_immediate(func, i1):
18         # equate the symbols
19         eff.add(s1)
20
21     # same check for the second input bit
22     if func(i1, not i2) == not o
23         and not is_immediate(func, i2):
24         # equate the symbols
25         eff.add(s2)
26
27     # returns an effective IFES
28     return eff
29 def compute_all_ifes(func: BoolFunction) -> list:
30     input1 = [0,1]
31     input2 = [0,1]
32     mappings = []
33
34     # all possible input permutations
35     for i1 in input1:
36         for i2 in input2:
37             eff = ifes(func, i1, i2)
38             mappings = mappings + [((i1, i2), eff)]
39
40     # mappings contains a list of all mappings
41     # with entries like
42     # (i1 , i2), {symbols}
43     return mappings
44
45 # arbitrary function to be analyzed
46 # for the purpose of finding its IFESs
47 def square_func(i1: bool, i2: bool) -> bool:
48    # example with OR
49     return i1 or i2
50
51 print(compute_all_ifes(square_func))
52 # [((0, 0), {1, 2, 3}), ((0, 1), {2, 3}), ((1, 0), {1,
   ↪ 3}), ((1, 1), {3})]
```

4.2.3 Injection-Symbol Propagation

Every output bit of an instruction may become an input bit of a subsequent instruction. Considering the IFESs of individual instructions, an IFES symbol like α or β set at an instruction output is again an IFES symbol at an instruction input of a subsequent instruction. Therefore, IFESs are combinable across instructions to describe the propagation of bit flips, not only within the instruction but also *globally* across the instructions of the whole data flow. The IFES mappings, like those presented in Section 4.2.2, are *transitive* due to the direct semantic connection between the output and input bits of subsequent instructions.

Figure 4.7 visualizes an example of such a transitive concatenation of IFES mappings, demonstrated by the bit 1 of the two registers `R0` and `R1` at the end of Listing 4.3b, a DFG segment in Figure 4.5.

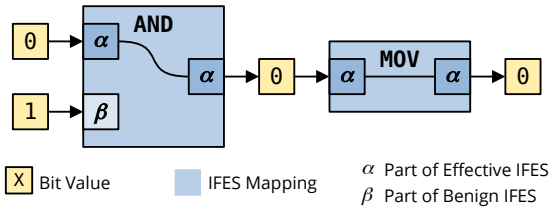


Figure 4.7 – Concatenation of Instruction-Local Fault-Equivalence Sets.
Figure 4.5 shows a segment of the DFG and the IFES mappings for the `AND` ($\beta \circ \alpha \rightarrow \alpha$) and MOV ($\alpha \rightarrow \alpha$) instructions, connected according to the data flow of the program from Listing 4.3b. A bit flip occurring from the *zero* at the `AND` input to the program's final output value has an equivalent effect. A bit flip of the *one* at the `AND` input is benign.

In the following instructions, the `AND` instruction executes at bit position 1, the calculation $1 \wedge 0 \rightarrow 0$, and its corresponding IFES mapping is $\beta \circ \alpha \rightarrow \alpha$. The output bit 1 of the `AND` instruction is then carried forward in the `MOV` instruction, resulting in the mapping $\alpha \rightarrow \alpha$. By concatenating both IFES mappings, the mapping for this bit becomes

$$1 \wedge 0 \xrightarrow{\texttt{AND}} 0 \xrightarrow{\texttt{MOV}} 0 \;\Rightarrow\; \beta \circ \alpha \xrightarrow{\texttt{AND}} \alpha \xrightarrow{\texttt{MOV}} \alpha$$

Any bit position marked with α will result in the same corrupted output when one of the bits experiences a bit flip. Consequently, all such bits in this example represent a transitive concatenation of IFES mappings, which I call *Data-Flow–Aware Fault-Equivalence Set (DFES)*.

To plan FI pilots in an FI campaign, *only* one FI is required for any position marked with α (`R1` bit 1 in t_2 and t_3, as well as `R0` bit 1 in t_4) in the FS. However, with DUP, the two read operations of the two instructions and using the final value in the system output result in *three* separate FESs, which must be injected separately in this short example, resulting in three necessary FI pilots. Section 4.3.2.1 will provide further details on the comparison between DFP and DUP and their relation to each other.

4.2.3.1 Initiation

The first step is introducing unique identifiers for each DFES to construct DFESs using the IFES mappings in the system's global scope. In the context of this dissertation, I refer to these identifiers as *injection symbols*. These symbols are potential identifiers for FI pilots executed in an FI campaign. Henceforth, when referring to *symbols*, it should be understood that I am explicitly referring to injection symbols.

A separate symbol is introduced for each bit that is reachable in the data flow. Figure 4.8 visualize this for the example with the instructions `AND` and `MOV` from the previous section. The upper bound for the maximum number of symbols in this example corresponds to the number of potentially necessary FIs in the naive approach $n_{\text{naive}} = 2 \cdot 4 \cdot 3 = 24$ to completely cover the FS, which corresponds to the size of the FS $|\mathscr{F}|$ (see Section 2.2.1 on page 33). Since the register `R0` and `R1` after `MOV` are not read or written after `AND`, they do not appear in the data flow. Thus, for this example, initially, there are 16 symbols `a`,...,`p`, one for each bit.

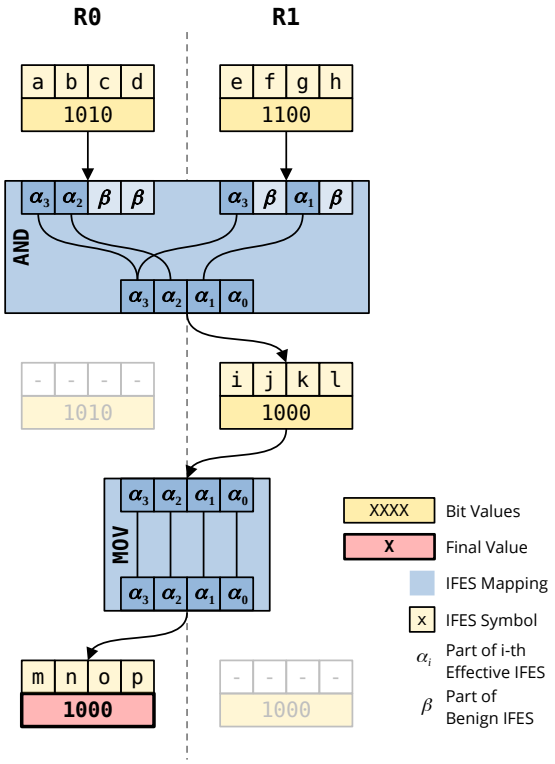


Figure 4.8 – Initialization of the Fault-Injection–Symbol Propagation.
This segment of the DFG illustrates the `AND` and `MOV` instructions along with their corresponding value nodes. Each instruction shows the various effective IFESs $\alpha_3, \ldots, \alpha_0$, as well as the benign `IFES` β. The edges within the instruction nodes represent the IFES mappings. Additionally, each value node contains the register values at a specific time and an injection symbol for every individual bit within the DFG. The DFP assigns a unique symbol to each bit in the DFG. In this particular segment, the DFG encompasses 16 symbols ranging from `a` to `p`. The concealed value nodes do not appear in the DFG itself but are part of the FS; thus, the bits associated with these nodes belong to the ineffective FS $\mathscr{I}$ when applying DFP.

4.2.3.2 Symbol Equalization

Now that all mapped symbols exist, the DFP *equalize* the symbols across the instructions. For now, let us examine the instructions separately. Equation 4.3 shows the IFES mappings for the `AND` instruction. The calculation process used IFES and equates injection symbols, which correspond to bits that can be assigned the same symbol. For instance, the symbol `i` overwrites the symbols `a` and `e`.

$$\begin{aligned}
1 \wedge 1 \to 1 \Rightarrow \alpha_3 \circ \alpha_3 \to \alpha_3 \quad & \Rightarrow a = e = i \\
0 \wedge 1 \to 0 \Rightarrow \alpha_2 \circ \beta \to \alpha_2 \quad & \Rightarrow b = j \qquad (\neq f) \\
1 \wedge 0 \to 0 \Rightarrow \beta \circ \alpha_1 \to \alpha_1 \quad & \Rightarrow g = k \qquad (\neq c) \\
0 \wedge 0 \to 0 \Rightarrow \beta \circ \beta \to \alpha_0 \quad & \Rightarrow l \qquad (\neq d \neq h)
\end{aligned} \tag{4.3}$$

Similarly, in Equation 4.4, it is also done for the `MOV` instruction.

$$\begin{aligned}
1 \to 1 \Rightarrow \alpha_3 \to \alpha_3 \quad & \Rightarrow l = p \\
0 \to 0 \Rightarrow \alpha_2 \to \alpha_2 \quad & \Rightarrow k = o \\
0 \to 0 \Rightarrow \alpha_1 \to \alpha_1 \quad & \Rightarrow j = h \\
0 \to 0 \Rightarrow \alpha_0 \to \alpha_0 \quad & \Rightarrow i = m
\end{aligned} \tag{4.4}$$

Equations can span across connected instructions, thus, the group of bits in the `DFG` that share the same symbol is considered part of a DFES:

$$\begin{aligned}
a = e = i = m \\
b = j = h \\
g = k = o \\
l = p
\end{aligned}$$

In the current implementation, the DFP performs a *backward breadth-first analysis* on the DFG in this step. In the beginning, for each value node in the DFG that does not have a successor node (meaning the value is no longer read), the IFES mappings are applied based on the instruction that produced this value. This process is repeated for the input value nodes of that instruction node until all value nodes are visited. Listing 4.2 provides the algorithmic sketch of this step, and Figure 4.9 visualizes the different states of the DFG. In the end, symbol equalization effectively reduces the number of symbols from 16 to just 4.

4.2.3.3 Conditions for Symbol Equalization

As demonstrated thus far, the propagation of symbols relies solely on the IFES mappings. However, two additional conditions determine whether the IFES mapping propagates a symbol. The first condition prevents establishing false equivalences where, for instance, a fault can interact with itself. Thus, it must be verified if *another* instruction is not reading the input value. Secondly, it is essential to ensure that a value becomes inaccessible before it can be used in subsequent instructions following a propagated fault. During this process, the DFP estimates the *lifetime* of the value, and based on this estimation, the symbols are either propagated or not.

Multiple Readers of a Value

Let us examine an example, which Figure 4.10 illustrates. Figure 4.10a shows the DFG, where a unique injection symbol is initially assigned to each bit. The IFES mapping for `XOR` is $\alpha \circ \alpha \to \alpha$, resulting in the equivalence of symbols to $b = c = d$. Similarly, the IFES mapping for `MOV` leads to the equivalence of symbols $a = c$. By concatenating the mappings shown in Section 4.2.3.2, the DFP infers that all symbols are equivalent: $a = b = c = d$.

To check this inference, let us examine Table 4.10b, where the bit values are displayed line by line at the corresponding symbol positions. Each column represents either the bit values in the absence of bit flips or the bit flips at the respective symbol positions, along with the resulting affected bits.

Listing 4.2 – Data-Flow Graph Backward Breadth-First Analysis with Instruction-Local Fault-Equivalence Set Mapping Application.
The function `apply_mapping` transfers symbols from output to input values based on the determined IFESs. The DFP computes these IFESs using the `ifes` function, an adapted version from Listing 4.1. This `ifes` function inputs value nodes with their corresponding values and symbols. It then checks each bit individually against the input bits of the instruction to determine if the corresponding input symbols exist within the corresponding IFES. Otherwise, the symbol belongs to a benign IFES and is `None`. The `propagate_symbols` function performs a backward breadth-first analysis. Initially, a queue gets all value nodes of the DFG that do not have an outgoing edge. For each of these value nodes, the `apply_mapping` function executes. Afterward, the input nodes of the instruction are added to the queue. This process repeats until the queue becomes empty, meaning the entire DFG has been traversed.

```python
def apply_mapping(i_node: InstructionNode) -> list:
    in_1, in_2, out = i_node.all

    for out_bit in range(0, WORD_SIZE):
        ifes = ifes(i_node.func, in_1, in_2)
        out_symbol = out.symbol[out_bit]

        # equalize when IFES contains symbol
        for in_bit in range(0, WORD_SIZE):
            if  in_1.symbol[in_bit] in ifes[out_bit]:
                in_1.symbol[in_bit] = out_symbol
            else:
                in_1.symbol[in_bit] = None # is
                ↪ benign
            if  in_2.symbol[in_bit] in ifes[out_bit]:
                in_2.symbol[in_bit] = out_symbol
            else:
                in_2.symbol[in_bit] = None # is
                ↪ benign
    return instruction.inputs
```

```python
def propagate_symbols(dfg: DataFlowGraph):
    # queue initialization
    queue = Queue()
    for value_node in dfg._all_values:
        if value_node.num_output == 0:
            queue.put(value_node)

    # traverse the whole DFG
    while not queue.empty():
        value_node = queue.get()
        # apply mappings for predecessor instruction
        inputs = apply_mapping(value_node.input)
        # add input nodes of instruction to queue
        queue.put(inputs)
```

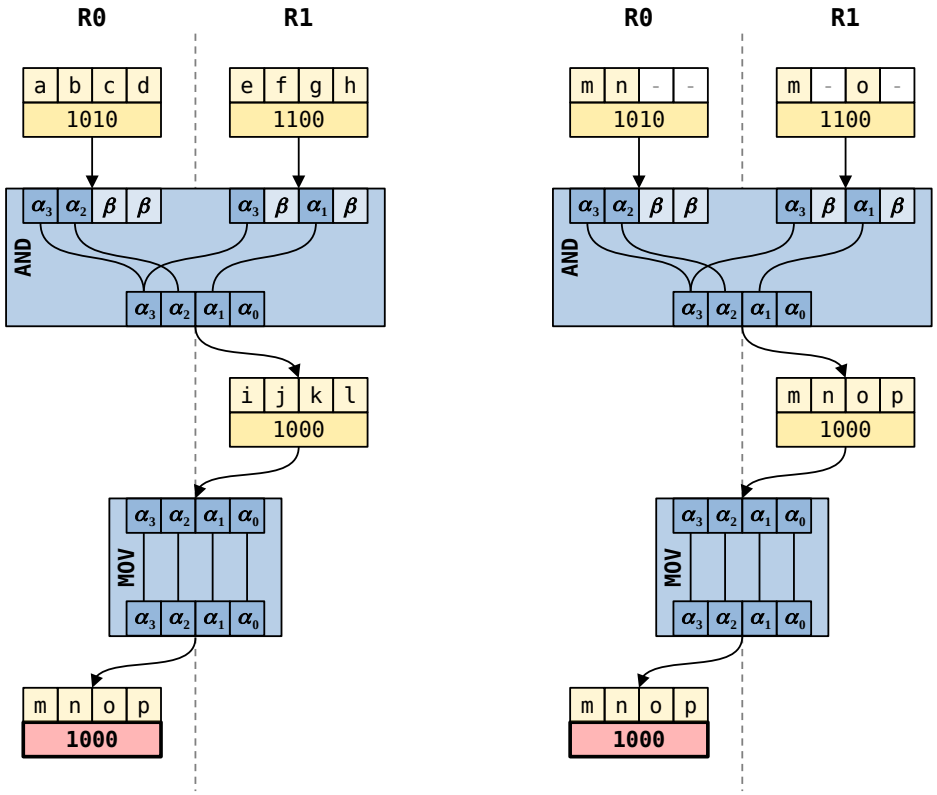
The DFP assigns the symbol `d` to the output bit of the graph. The equation $a = b = c = d$ implies that bit flips at the bits of all symbols in the graph should yield the same final result. However, Table 4.10b shows that flipping the bit represented by `a` does not produce the same result at the end as flipping the other symbols. Therefore, the assumption $a = b = c = d$ is *incorrect*.

When the bit represented by `a` flips, the `MOV` instruction copies it to another location, and the bit flip remains in its original location. The `XOR` instruction semantically reads the same bit flip in both inputs, thus interacting with itself. The symbol `a` cannot be equalized with $b = c = d$ since this violate of the single-fault assumption of the FM (see Section 2.2.4 on page 41). If the bit at `a` flips from *zero* to *one*, the result bit remains *zero*, whereas an injection in a bit with one of the remaining symbols provokes a result of *one*, which highlights the first *condition* for symbol equalization:

A symbol of a bit must only be propagated if the current instruction is the only (interpreting) reader of this bit.

To account for multiple reads of a value in the data structure of the DFG, I introduce the *epsilon transition*. The epsilon transition is a *virtual* instruction node that does not propagate symbols. An epsilon transition is added whenever an instruction reads a value, duplicating the corresponding value node. From now on, this node represents the most recent *instance* of the value in the system.

Figure 4.11 shows the creation of chains of epsilon transitions. These chains capture different instances of reading a value and the varied propagation behaviors of the bit flips. This distinction is crucial for understanding the propagation of a bit flip, as it can affect different instructions without propagating the associated symbols.



(a) Initialitation

(b) Symbol Equalization

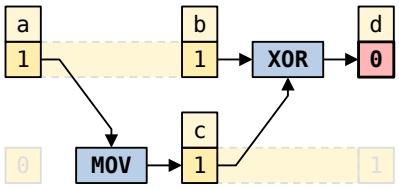
Figure 4.9 – Fault-Injection–Symbol Propagation.
The meaning of the illustration's individual components is the same as that of Figure 4.8. Figure 4.9a represents the initial state of the DFG. Using the IFES mappings, symbol equalizations are performed, as depicted in Equation 4.3 and Equation 4.4, and applied to the DFG, as seen in Figure 4.9b. This process effectively reduces the number of distinct symbols required for full coverage of the DFG from 16 to only 4 necessary FIs.

Figure 4.12 illustrates the expansion of the initial DFG (from Figure 4.5) by epsilon transitions. In this particular case, no chains of epsilon transitions exist. However, such chains can be present when dealing with global constants in the code, for instance. Including epsilon transitions allows for a more comprehensive representation of the data flow and facilitates the analysis of different propagation behaviors in the presence of bit flips.

Value Lifetime

Another crucial condition for symbol equalization is the consideration of the *lifetime of values*, which involves dead values that are only read in diverging execution flows *after* the occurrence of a bit flip. The code example presented in Listing 4.3 is a simple yet comprehensive illustration, highlighting the necessity of accounting for value lifetimes in symbol equalization.

In the correct execution of this code, the dereference instruction in line 4 is not executed, meaning there is no corresponding instruction in the data flow that acts as a second reader for the `ptr` variable.



(a) Data-Flow Graph with Multiple Readers of Bit at `a`

	none	a	b	c	d
a	1	0	1	1	1
b	1	0	0	1	1
c	1	0	1	0	1
d	0	0	1	1	1

(b) Bits' Final Values Under Various Bit Flips Scenarios

Figure 4.10 – Mismatch of Symbol Equalization: Impact of Multiple Value Readers.
When a bit in Figure 4.10a flips at symbol `a`, the flipped bit is copied to symbol `c`'s location while also remaining at its original location, which leads to an interaction of the bit flip with itself, as it affects the inputs of the `XOR` instruction.
Table 4.10b represents five different bit flip scenarios in each column: no bit flip, bit flip at symbol `a`, ..., bit flip at symbol `d`. Each row represents the final bit values resulting from these scenarios. Based on the IFES mapping rules outlined in Section 4.2.2, the symbol equivalence $a = b = c = d$ holds. However, in this example, the bit flips lead to different final bit values (highlighted cells in Table 4.10b), indicating an incorrect assumption of symbol equivalence.

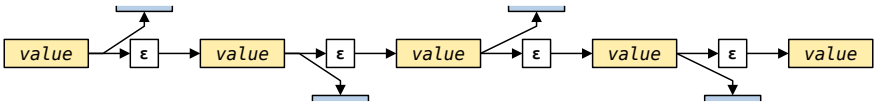


Figure 4.11 – Epsilon-Transition Chain.
When a value is read multiple times, a bit flip in that value can have different effects at different points in time between these read accesses. To differentiate the value at different instruction interpretations, the DFG includes epsilon transitions (ϵ), creating a chain of transitions. In this particular example, the value is used four times as an input for arbitrary instructions, leading to the creation of this chain.

Listing 4.3 – Exemplary Code for Highlighting the Impact of a Value's Lifetime.
The code assigns a null pointer stored in `ptr` to `tmp`, resulting in a subsequent operation that makes the if condition in line 3 false. However, if bit flips occur in either `ptr` or `tmp` at the beginning of this code, the if statement becomes true, and a different failure class arises, depending on bit flips in `ptr` or `tmp`. A bit flip in `ptr` results in dereferencing `ptr`, whereas a bit flip in `tmp` leads to a trap since `ptr` remains a null pointer and cannot be dereferenced.

```
1  void *ptr = NULL;
2  void *tmp = ptr;
3  if (tmp != NULL) {
4      *ptr;
5  }
```

However, it is essential to note that bit flips in `ptr` and `tmp` are not equivalent, as they can result in different failure classes. For instance, if a bit flip occurs in `ptr` at the beginning and coincidentally transforms it into a valid pointer, the subsequent dereference operation will not cause a trap, and the program will terminate successfully. Therefore, even though the assignment operation in line 2 is the only reader in the data flow, symbol equalization cannot be applied.

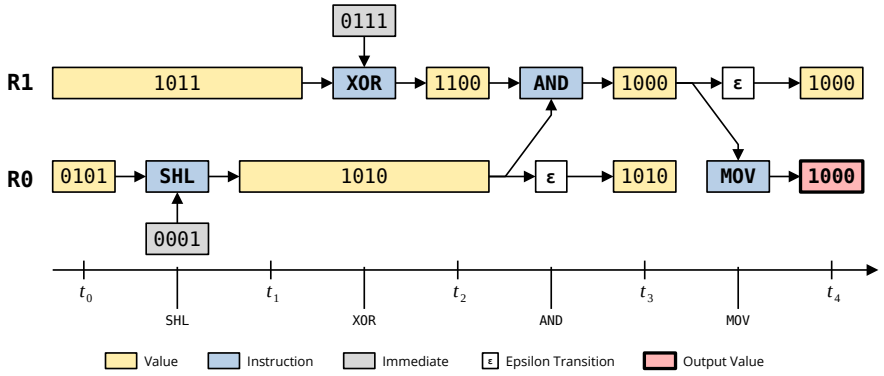


Figure 4.12 – Data-Flow Graph with Epsilon Transitions.
This DFG is the same as in Figure 4.5 but includes epsilon transitions additionally. Each epsilon transition represents a new value instance after an instruction reads it. These transitions help to differentiate the value at different points in time between read accesses within the data flow graph.

Referring to this example and as a result of it, I introduce the second condition that must apply to the propagation of symbols:

A symbol of a bit must only be propagated if the current input value bit of the (interpreting) instruction has become globally inaccessible.

To do this, I check that the lifetime of the input value is smaller (or equal) to the end time of the output value node. This condition guarantees that the input value is no longer read or used in any instruction after its lifetime.

It must be ensured that even if the bit flip results in a deviating control- or data flow, it cannot interact with itself and violate the single-fault assumption. To implement this condition, I use the overwrite time of the value after the last epsilon node as a conservative estimation of its lifetime. Regarding the epsilon chains in a DFG, as shown in Figure 4.11, it is always the last value node in that epsilon chain. By comparing the overwrite time of the input value with the end time of the output value node, it is possible to determine if the input value has indeed become inaccessible before the symbol propagates. This approach provides a reliable means of preventing potential interactions or conflicts between multiple faults.

However, in some situations, it is possible to use a shorter lifetime approximation if it is possible to demonstrate that a value becomes inaccessible earlier. Values stored in registers end their lifetime if no instruction is issued after the output value node reads the value before it is overwritten. For example, in Figure 4.4, the value `1000` in register `R1` is present until after time t_4, but it becomes inaccessible after t_3 since no instruction in between uses `R0` as an input.

4.2.4 Fault-Injection–Campaign Planning

At this stage, let me summarize what we have archived so far:

1. A DFG was created to represent the *data flow* of the analyzed program (see Section 4.2.1).
2. IFESs were determined, which are derived based on their specific IFES *mappings* regarding the appropriate instruction (see Section 4.2.2).

3. To obtain a global perspective, the IFESs were concatenated to *propagate injection symbols* across instructions (see Section 4.2.3).

4. The propagation of symbols was carried out based on the corresponding IFES mappings, subject to *two conditions* (see Section 4.2.3.3): (a) the input values must not be interpreted by any other instruction (represented by extending the DFG with epsilon transitions), and (b) it should be determined whether the value becomes inaccessible (determined by the lifetime of the most recent value instance).

After completing all these steps, we reach the state of the DFG illustrated in Figure 4.9b.[34] These steps provide a comprehensive understanding of the analyzed program's data flow and symbol propagation, setting the foundation for the last step of the DFP: *Fault-Injection–Campaign Planning*. In this final step, the DFP uses the DFG with propagated symbols to determine *pilots* for an FI campaign to ensure that the generated pilots cover the entire FS and produce results to make a statement about the system's reliability precisely.

Initially, each symbol represents a point in the FS and serves as a potential pilot for the FI campaign. The symbols are propagated through symbol equalization by concatenating IFES mappings, ensuring equivalent system behavior in the presence of a bit flip. As a result, each distinct symbol corresponds to a single *Data-Flow–Aware Fault-Equivalence Set (DFES)*. Equal to the FES from DUP (see Section 2.3.2.1 on page 47), a single representative or pilot is chosen for each DFES. Therefore, the number of pilots is the number of DFES or unique symbols in the DFG.

However, the DFP performs a backward breadth-first analysis through the DFG a *second* time to determine the pilots. The DFP needs to perform two complete traversals because the queue's last instruction node conclusively determines the symbols' propagation with the final symbol equalization check. It is unclear when propagation should stop or continue during the first run. Therefore, the second run must ensure that all necessary symbol equalizations occur accurately.

Listing 4.4 shows the exemplary code for determining the pilots. This planning step performs a breadth-first search analysis, similar to the analysis in Listing 4.2. This code keeps track of the distinct symbols already read in the DFG to create *only one* pilot per unique symbol, represented by the structure `symbol_to_pilot`. Additionally, it records the set of pilots in the variable `pilots` during the traverse of the DFG. The current value node is an instruction's output node, and the instruction's input values are added to the queue. This process continues until the queue is empty (i.e., the algorithm traversed the entire DFG).

The number of covered bits of the corresponding DFES represents the *pilot's weight*. To determine the pilot's weight, the DFP counts how often each symbol appears in the DFG, which is the weight $w(p_i)$ of each corresponding pilot p_i with the help of the mapping $s_i \rightarrow p_i$, which keeps track which symbol s_i is represented by which pilot p_i.

With the resulting set of pilots P_{DFP} with n_{DFP} pilots and its corresponding weights, the FI campaign is ready for conduction.

4.3 Evaluation

This section presents the evaluation results of the implemented Data-Flow–Sensitive Fault-Space Pruning in FAIL*. To begin, I present the *distributions* of the instructions in my benchmark portfolio,

[34] The two conditions mentioned from Section 4.2.3.3 are also included in Figure 4.9b. The value in register `R1`, before the `MOV` instruction, is not interpreted by any other instruction besides this `MOV`. Additionally, this value is not interpreted at any other time, indicating that it is outside its value lifetime. The conditions mentioned earlier also hold for all other instances in Figure 4.9b.

Listing 4.4 – Examplary Code for the Fault-Injection–Planning Step of the Data-Flow Pruning.
This code represents the DFP's planning step. During this second traverse through the DFG, the DFP identifies the pilots set by recognizing unique symbols and defining a pilot for each. The structure `pilots` records these pilots as well as the map of symbols to pilots `symbol_to_pilot`. Additionally, the DFP computes the weights of the pilots in the structure `weights`, corresponding to the counts of each symbol.

```
1   def planning(dfg: DataFlowGraph) -> set, dict, dict:
2       # queue initialization
3       queue = Queue()
4       for value_node in dfg._all_values:
5           if value_node.num_output == 0:
6               queue.put(value_node)
7
8       pilots = set()
9       symbol_to_pilot = dict()
10      weights = dict()
11      # traverse whole DFG
12      while not queue.empty():
13          value_node = queue.get()
14          for bit in range(0, WORD_SIZE):
15              # check if symbol is not benign and is not already used for pilot determination
16              # then create pilot, recods pilot's symbol and initalize weight
17              # else increase weight for that symbol
18              symbol = value_node.symbol[bit]
19              if symbol is not None and symbol not in symbol_to_pilot:
20                  # creates a pilot as a tuple (time, location bit)
21                  pilot = (value_node.time, value_node.data_address + bit)
22                  pilots.add(pilot)
23                  symbol_to_pilot[symbol] = pilot
24                  weights[symbol] = 1
25              else:
26                  weights[symbol] = weights[symbol] + 1
27
28          # get the instruction node of the value
29          instr_node = value_node.input
30          # get the input values of the instruction
31          inputs = instr_node.inputs
32          # add input nodes of instruction to queue
33          queue.put(inputs)
34
35      return pilots, symbol_to_pilot, weights
```

focusing on the instructions selected for the IFES mappings (see Section 4.2.2.2 in this dissertation. Next, I discuss the *validation* of the DFP and its relationship to the DUP. Finally, I present the evaluation *results*, divided into two aspects: the overhead of the approach and the effectiveness of the outcomes regarding the number of determined pilots.

4.3.1 Distribution of Instructions in the Benchmarks

To understand the instruction patterns in the benchmarks portfolio (see Section 3.2.3.1), I analyzed the distributions of instructions using the program's golden run and its objdumps. I focused on instructions for which I have implemented IFES mappings in FAIL*.

Table 4.1 provides an overview of the instruction counts (#Instr.) and their distributions across the benchmarks, expressed in percentages.

Figure 4.13 visualizes this information as a stacked bar plot. Since the DFP relies on instruction-local equivalences, the distribution of instruction types is an vital characteristic of the benchmarks. The handled instruction types (`MOV`, `ADD`, `AND`, `OR`, `XOR`) are categorized, and the label `other` captures all instructions that the DFP treats as opaque.

The analysis reveals that the DFP considers between 26 percent (μLSUM) and 68 percent (miSHA) of the instructions for propagation. `MOV` instructions are the most prevalent among the instruction

		Per Instruction Type [%]					
	#Instr.	AND	ADD	MOV	OR	XOR	other
miBC	54 434	12.59	14.25	12.58	0.00	0.75	59.83
miBFD	55 647	5.41	7.86	30.22	1.35	13.11	42.05
miBFE	54 951	5.51	7.97	28.87	1.38	13.37	42.91
miQS	45 774	0.03	8.44	47.15	0.57	0.32	43.49
miRDD	70 824	0.01	1.11	37.90	0.00	22.23	38.76
miRDE	70 450	0.02	0.84	37.79	0.00	22.11	39.24
miSHA	40 647	3.16	9.18	42.88	2.37	10.77	31.65
μFIB	2 093	0.00	13.76	10.32	0.00	2.63	73.29
μLSUM	54	0.00	16.67	9.26	0.00	0.00	74.07
μMIX	90	8.99	10.11	17.98	8.99	8.99	44.94
μQSI	800	0.13	7.46	51.35	0.00	0.00	41.06
μQSR	804	0.00	7.56	36.97	0.00	0.00	55.46

Table 4.1 – Benchmark Portfolio's Instruction Distribution.
This table displays the distribution of instructions in the benchmark portfolio from Section 3.2.3.1. The DFP contains IFES mappings for highlighted instructions, categorizing all other instructions not considered by the DFP under the label `other`.

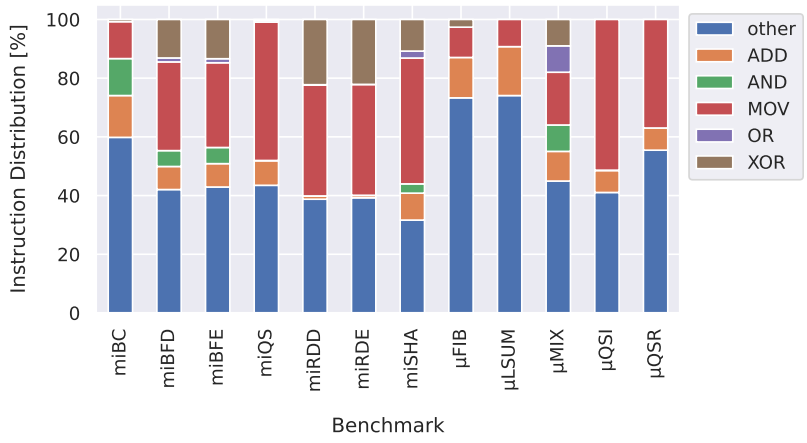


Figure 4.13 – Benchmark Portfolio's Instruction Distribution Stacked Bar Plot.
This figure presents the visual representation of the instruction distribution across the benchmark portfolio with the data from Table 4.1.

types, accounting for up to 51 percent (μQSI) of the instructions. This high percentage highlights the potential for the most straightforward IFES mapping of the DFP technique. In two benchmarks (μLSUM and μQSR), the bit-wise instructions (`AND`, `OR`, `XOR`) do not occur at all. On the other hand, in specific benchmarks like μMIX, the bit-wise instructions make up a more significant part of the instructions (27 percent) than others.

Overall, the distribution of instruction types provides insights into the pruning potential of the DFP technique across the dissertation's benchmark portfolio, with `MOV` and `XOR` instructions showing promising prospects.

4.3.2 Validation of the Data-Flow Pruning

At this point, I will describe the validation of the DFP to ensure the sanity of the evaluation results. I will outline the features and differences between DUP and DFP and then describe the approach used to validate the DFP's correctness.

4.3.2.1 Comparison of Def/Use Pruning and Data-Flow Pruning

The DUP focuses on the read- and write accesses of instructions in the data flow of the SUT. The DFG also represents these accesses, which capture each instruction's input and output values and their corresponding execution times. By leveraging the DFG, the DFP can precisely map the read and write accesses used in the DUP, thus accurately *reproducing* the DUP's FESs.

Figure 4.14 presents the running example of the previous sections and shows the application of both DUP and DFP on the same segment of the FS. The left side shows the FS with the application of DUP, and the right side presents the FS with the application of DFP, including the IFES mappings for the `AND` and `MOV` instructions. As a reminder, the value in register `R0` represents the system's output. Through the use of DFP, the number of required pilots decreases significantly compared to DUP. In this example, the number of pilots decreases from 16 in DUP to only four in DFP. The pilots in DFP are weighted based on the effective FS coverage achieved after executing DFP, as explained in Section 4.2.4.

Taking the `MOV` instruction as an example, the eight FESs of DUP are *unified* into four DFESs using the IFES mapping for `MOV`. Similarly, for the `AND` instruction, the `IFES` mapping combines individual FESs, creating larger sets of equivalences and uniting existing FESs from DUP. Notably, in the case of the `AND` instruction, certain bits in the `FS` for the `FI` campaign are entirely disregarded due to the analysis of `AND` semantics, which reveals that bit flips in certain bits, such as bit 0 in `R0` at time t_2, are benign and can be ignored, leading to a *reduction* in FESs compared to DUP.

Without IFES mappings in the DFP, the edges on the right side of Figure 4.14 vanish, and the information regarding benign bit flips disappears. Consequently, only the details about the read and write accesses from the DFG would remain, which implies that if no IFES mappings are activated or implemented in the DFP, the DFP semantically falls back to the semantics of the DUP with equivalent determined pilots and its weights. With the incorporation of IFES mappings and the propagation of the symbols, the DFP emerges as an *enhanced version* of the DUP.

4.3.2.2 Validation Procedure

For the validation of the DFP, I compare the results of DUP and DFP methods. The goal is to demonstrate that the DFP calculates the same complete and precise failure classifications for all benchmarks. FAIL* records the failure classes using the equivalent planned pilots for identical FI campaign settings. A thorough per-FI comparison ensures that both methods yield identical failure classifications for all the pilots. Overall benchmarks, DUP covers $1.3 \cdot 10^{10}$ faults using $2.21 \cdot 10^{7}$ pilots. When the IFES mappings of the DFP are deactivated, the number of pilots is the same $n_{\text{DUP}} = n_{\text{DFP}}$, as expected. However, with the activated IFES mappings, the DFP determines $1.93 \cdot 10^{7}$ pilots. Importantly, in all cases, both methods agree on the failure classes of the covered FSs. At the same time, the DFP can cover the FSs of the benchmarks with fewer pilots, demonstrating its effectiveness in reducing the number of injections while maintaining accurate failure classification coverage.

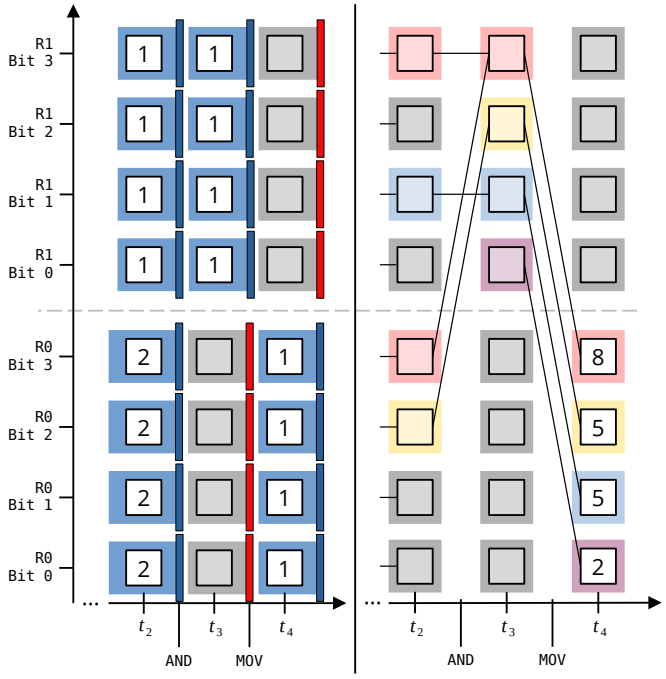


Figure 4.14 – Comparison of a Fault Space After Applying Def/Use Pruning and Data-Flow Pruning.
The segment of the running example's FS is shown twice, once after applying DUP and once after applying DFP. The DUP determines 16 FES with weights of 1 or 2, and the DFP creates only four FESs with weights of at least 2 and up to 8. The DFP combines distinct FESs of the DUP and ignores FESs deemed benign. This figure also explicitly shows the transition from DUP's one-dimensional FESs to DFP's two-dimensional FESs; the DUP creates FESs only regarding the temporal dimension and the DFP incorporates the location dimension of the FS.

4.3.3 Results

This section focuses on the evaluation *results*, explicitly examining DFP's *efficiency* and *effectiveness*. The evaluation measures improvements in both efficiency and effectiveness as defined in Section 3.2.1.3 from page 63.

4.3.3.1 Data-Flow–Pruning Efficiency: Overhead

First, let us examine how *efficiently* the DFP comes to a result. The calculation of pilots using the DFP for the Micro benchmarks took at most 1.1 seconds, whereas for the MiBench benchmarks, it ranges from 2.5 minutes (miBFD) to 24 minutes (miRDD). In comparison, the DUP approach took no longer than 11 seconds for all benchmarks, with the longest pruning time observed for miRDE. These relatively long pruning times result from the current prototypical implementation, which visits value nodes multiple times. The DFP scales linearly with the number of instruction and value nodes, as the ISA limits the number of DFG predecessors. Overall, each node needs to be visited only twice.

Figure 4.15 illustrates the DFP's execution times for the MiBench benchmarks, segmented into the four individual steps as described in sections Section 4.2.1 to Section 4.2.4. The predominant step is symbol propagation, which significantly contributes to the longer runtimes of the DFP. Optimization

in the processing of symbol propagation could improve this prototypical implementation. Other steps exhibit lesser significance in the overall context: DFG creation consumes a maximum of 33 seconds (miRDD), IFESs calculation takes not more than 1.5 seconds (miRDE), and pilot planning requires a maximum of 44 seconds (miRDD).

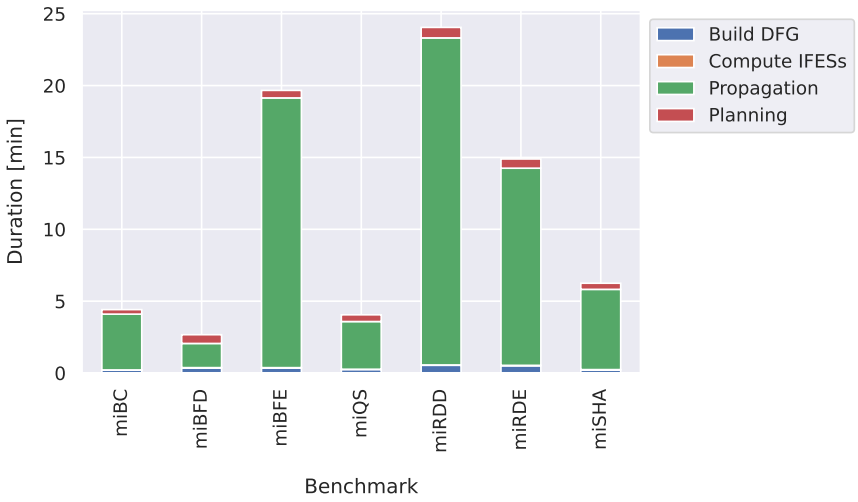


Figure 4.15 – Data-Flow–Pruning Single Step's Overhead.
This figure illustrates the DFP's execution times across the MiBench benchmarks. Each box represents a specific step of the DFP process. The symbol propagation step notably dominates the DFP's runtime compared to other steps.

Despite the prototypical implementation of DFP's symbol-propagation step, the DFP still achieves significant end-to-end campaign-runtime savings for all MiBench and Micro benchmarks when comparing the pruning time to the overall compute-time demand of the FI campaign. For example, in the case of miRDD, DFP executed single-threaded for 0.4 hours to reduce the number of injections by 13.1 percent. If the FI campaign executes the DUP pilots on a single-threaded machine, it will take 188 hours, which I can reduce by 24 hours using DFP. Even when considering the pruning overhead, using DFP pilots save more than 23 hours.

4.3.3.2 Data-Flow–Pruning Effectiveness: Reduction of the Number of Pilots

In this section, I present the evaluation results regarding the *effectiveness* of the DFP. I examine its overall effectiveness (i.e., the reduced number of required pilots; refer to Section 3.2.1.3 on page 61) and sketch two additional characterizations of the DFP pilots. Finally, I group the results based on the resulting failure classes of the FIs.

Overall Reduction

In the following analysis, I quantify the reduction in pilot count achieved by the DFP compared to the DUP and present the results in Table 4.2. The columns related to Def/Use Pruning display the number of required pilot injections (n_{DUP}) and the average weight ($\overline{w(p_i)}$) of a single pilot in the context of the entire FI campaign. A higher per-pilot weight indicates that the pruning method can yield more extrapolated results per performed FI. In the case of DFP, I provide the same information

along with additional columns showcasing the percentage improvements (Δ) over DUP. In the last column group, I provide additional information about the planned DFP pilots: Within the pilots, *loc* percent of the corresponding DFESs cover multiple fault locations in the FS,[35] and *val* percent of the corresponding DFESs cover more than one value node in the DFG, which includes the pilots referenced by *loc*. For DUP, both metrics are zero.

	Def/Use Pruning		Data-Flow Pruning				DFP Pilots	
	n_{DUP} [10^4]	$\overline{w(p_i)}$	n_{DFP} [10^4]	$\overline{w(p_i)}$	Δn [%]	Δw [%]	loc [%]	val [%]
miBC	222.40	31.63	181.43	38.77	-18.42	+22.58	1.56	3.85
miBFD	331.38	571.79	295.95	640.24	-10.69	+11.97	2.77	4.92
miBFE	326.93	575.30	292.97	641.98	-10.39	+11.59	2.73	4.67
miQS	270.58	600.15	234.31	693.05	-13.40	+15.48	3.88	4.62
miRDD	397.60	881.84	345.37	1015.18	-13.13	+15.12	1.56	4.48
miRDE	397.99	868.77	351.90	982.55	-11.58	+13.10	1.89	4.53
miSHA	252.79	95.98	219.74	110.42	-13.07	+15.04	5.41	9.02
μFIB	8.87	12.98	7.56	15.24	-14.78	+17.35	2.35	7.51
μLSUM	0.26	6.35	0.26	6.35	0.00	+0.00	0.00	0.00
μMIX	0.45	6.61	0.40	7.49	-11.83	+13.42	0.00	6.38
μQSR	1.27	14.38	1.22	15.04	-4.36	+4.56	1.84	4.32
μQSI	4.23	28.43	3.88	30.96	-8.18	+8.90	5.71	7.71

Table 4.2 – Comparison of Def/Use Pruning and Data-Flow Pruning.
This table provides a quantitative comparison between DUP and DFP. The first two groups of columns display the number of calculated pilots (n_{DUP} and n_{DFP}) and the average weights of the pilots ($\overline{w(p_i)}$) for DUP and DFP, respectively. The highlighted columns in the DFP group represent the percentage difference in the number of pilots (Δn) and the average weight of the pilots (Δw). In the *DFP Pilots* columns, *loc* indicates the proportion of DFP pilots that cover more than one location in the FS with their DFESs, and *val* indicates the proportion of DFP pilots that cover more than one value node in the DFG.

The most crucial aspect of Table 4.2 is the percentage reduction in the required FIs (Δn). This column explicitly represents the effectiveness described in Section 3.2.1.3. When examining the MiBench benchmarks, we observe reductions ranging from 10.39 percent (miBFE) to 18.42 percent (miBC). With more equivalent classified faults, the DFP requires fewer injections while maintaining the same complete coverage of the FS as the DUP. Consequently, this leads to an increased weight per injection.

For the micro benchmarks, we observe reductions (excluding μLSUM) ranging from 4.36 percent (μQSR) to 14.78 percent (μFIB), accompanied by the necessary weight adjustments. However, in the case of μLSUM, the DFP cannot reduce the number of injections compared to the DUP. This outcome is not surprising considering the structure of this benchmark, which solely accumulates an array of integer values into a register. The array elements are not overwritten (maximal lifetimes), and the program does not perform immediate calculations on the accumulator register. Nevertheless, in such cases, the DFP gracefully degrades to the DUP.

Next, let us delve into the characterization of the planned DFP pilots. In the last column group, we observe that a relatively small percentage (< 5.1% for miSHA) of pilots cover more than a single fault location or more than a single value node (< 9.2% for miSHA). All other pilots represent the

[35] As a reminder, for DUP, there is always only one location in the FS per FES.

same set of faults as planned with DUP. This finding suggests that only a few DFP pilots contribute to the reductions and indicates that larger equivalence sets occur when propagation is possible. On average, these multi-value–node pilots span up to 6.9 value nodes.

Reduction per Failure Class

Next, I consider whether DFP is more likely to combine faults that result in certain failure classifications. Table 4.3 shows how the percentual change in the number of pilots leads to a specific failure class.

	OK	SDC	TIME	TRAP
miBC	-1.45	-42.00	0	-4.69
miBFD	0	0	-0.32	-12.04
miBFE	0	0	-0.36	-11.12
miQS	-25.96	-2.86	-0.15	-1.12
miRDD	0	0	0	-15.25
miRDE	0	0	0	-13.30
miSHA	0	-22.38	-0.04	-1.35
μFIB	-0.10	-38.72	-0.73	-27.68
μLSUM	0	0	0	0
μMIX	-27.66	-3.39	0	-4.16
μQSR	0	-5.59	-7.41	-11.32
μQSI	0	-15.03	0	-4.52

Table 4.3 – Pilot Reduction per Failure Class [%].
This table presents the data in the highlighted columns from Table 4.2 and groups it based on the failure classes of the executed FIs.

Across all benchmarks, SDC and trap failure classes exhibit more significant reductions, whereas other classes vary widely. This observation concludes that fine-grained data flows primarily influence the DFP's equivalence propagation. In summary, compared to the DUP, the DFP achieves reductions of up to 18 percent without compromising completeness or precision. Even in its prototype implementation, it significantly accelerates the overall campaign speed.

4.4 Summary of Data-Flow–Sensitive Fault-Space Pruning

To summarize, FAIL* is extended by the *Data-Flow–Sensitive Fault Space Pruning* [▷PDL21], DFP for short. The DFP is a semantic extension of the well-known *Def/Use Pruning (DUP)*; DFESs unify the one-dimensional FESs of the DUP *across locations* and are potentially *two-dimensional* regarding the FS (visualized in Figure 4.14). Compared to the DUP, the DFP considers the program's *data flow* to be analyzed instead of only considering the read and write accesses in registers and memory. The DFP is *instruction-aware* and *value-aware*, which includes the semantics of the instructions used and the input and output values for the analysis. The DFP uses a *Data-Flow Graph (DFG)* as the primary data structure, which reflects the program to be analyzed. Initially, the DFP defines an *injection symbol* for each bit in the DFG, which serves as a label for a potential pilot for the ongoing FI campaign. For each instruction of the DFG and based on its *instruction semantics*, an *Instruction-Local Fault-Equivalence Set (IFES)* mapping can be developed for the DFP, dividing the input and output

bits into *effective* and *benign* IFES. The DFP concatenates IFES mappings as specified in the structure of the DFG so that the injection symbols of an effective IFES are propagated using a DFG *breadth-first search analysis*, resulting in larger FESs compared to the DUP, the *Data-Flow–Aware Fault-Equivalence Set (DFES)*. Finally, each injection symbol remaining after the propagation step represents a *pilot*.

For a frequently occurring selection of instructions in the benchmark portfolio of this dissertation, namely `MOV`, `AND`, `OR`, `XOR` as well as `ADD`, IFES mappings were developed and the entire process for the DFP was implemented and evaluated in FAIL*.

For a frequently occurring selection of instructions in this dissertation's benchmark portfolio, namely `MOV`, `AND`, `OR`, `XOR`, and `ADD`, I developed IFES mappings. The entire DFP process is implemented and evaluated in FAIL*. A DFP effectiveness of up to 18.42 percent is measurable for the given benchmarks. The DFP effectively identifies DFESs that result in SDC and trap failure classes, indicating its firm reliance on fine-granular data flows. All pilots that emerge are to be used to complete and precisely cover the FS in an FI campaign. If no IFES mapping exists, the DFP generates pilots equivalent to the DUP so that the DFP can be seen as an *enhancement* of the DUP.

The code and evaluation data of the prototype implementation are publically accessible on Zenodo.

TRY IT OUT: Data-Flow–Sensitive Fault Space Pruning in FAIL*

The DFP can be compiled and executed seamlessly with an executable FAIL* installation with a Capstone Assembler. The source code of the DFP and its evaluation data is available here:

`https://doi.org/10.5281/zenodo.4698901`

5

Program-Structure–Guided Approximation of Fault Spaces

> I was trying to do too many things at the same time, which is my nature. But I was enjoying it, and I still do enjoy it.
>
> JAMES MARSHALL "JIMI" HENDRIX (1942–1970)

In this chapter, I present the work of my paper *Program-Structure–Guided Approximation of Fault Spaces*, which leverages the control-flow structure of a program, mainly jump instructions, in the fault-free execution trace to define FSRs. This approach focuses on injecting faults into data flows that cross the boundaries of FSRs, as these data flows transfer computation results from one region to another. I reduced the number of required pilots by assigning appropriate weights to each region and the corresponding injected faults, while fully covering the entire FS. After extensively exploring the data flow in the previous chapter to accelerate an FI campaign, this chapter focuses on the *control flow* and its utilization in reducing the number of required pilots for achieving full FS coverage in an FI campaign.

To understand this method, I will first describe the concept of program structure and its application in the context of this dissertation. I explore how program structure guides FSR creation and highlight its specific instances of FSRs. The FSR concept is integrated into FAIL*, and a comprehensive evaluation is conducted to assess the effectiveness of this method in reducing the number of required pilots and the locality of the obtained results. I implemented the FSR concept into FAIL* and conducted a comprehensive evaluation to assess its effectiveness in reducing the number of required pilots and the locality of the obtained results.

I presented the FSR [▷Pus+19] approach at the PRDC conference in 2019 and am the paper's primary author. My work included conceptualization, FSR implementation, benchmark design, experiment conduction, and paper writing.

5.1 Program Control Flow in the Context of This Dissertation

To begin, I will provide a general explanation of the control-flow concept within the scope of this dissertation and sketch the construction of control-flow graphs. The nodes in such a graph, known as *Basic Blocks (BBs)*, will be described in more detail throughout this section. Specifically, I will differentiate between static and *dynamic BBs* and present the implementation of dynamic BBs determination in FAIL*, used in the chapter's FI acceleration method.

5.1.1 Control Flow: Order and Execution of Instructions

A *control flow* is the order in which a computer program executes instructions or statements. It governs the execution flow as the program traverses various sections, including loops, conditionals, function calls, and other control structures [ASU86]. The control flow is a fundamental concept in computer programming that governs the order of instruction execution within a program. It is a crucial factor in program comprehension, debugging, and optimization. One common approach to analyzing control flow is identifying and studying *Basic Blocks (BBs)*, which provide a structured view of the program's execution flow.

BBs are consecutive instruction sequences within a program that start at a single entry point and end at a single exit point without any jump instructions. They serve as fundamental units of analysis in compiler design and optimization, playing a crucial role in tasks such as code generation, program analysis, and performance tuning. BBs provide a structured representation of program control flow, allowing for efficient program code analysis and transformation [ACK70].

Listing 5.1 displays a snippet of the objdump for the μQSI benchmark (see Section 3.2.3.1 on page 67). Each unconditional (`JMP`) or conditional jump instruction (e.g., `JL`, `JS`) marks the *end* of a BB, while the jump addresses define the *start* of a new BB. In the example provided, BBs `A` to `G` exist. Notably, BB `B` is split into `B1` and `B2`. Although `B1` and `B2` both end with the same jump instruction, they are accessed by different instructions (see lines 51 and 55). `B1` seamlessly transitions into the subsequent BB `B2`, ensuring a smooth and uninterrupted execution flow. Therefore, `B1` and `B2` are distinct BBs in this case.

BBs have fixed start and end instructions, allowing them to function individually within a program and exhibit modularity. The execution of a program consists of concatenating these BBs. A *Control-Flow Graph (CFG)* represents a program's possible BBs order. Figure 5.1 shows the code example from Listing 5.1 and its BBs as a CFG. Each BB serves as a node in the graph representation. Directed edges represent the transitions between the BBs. These transitions connect the last instruction of one BB to the first instruction of another BB. There are three types of transitions: (1) A conditional taken jump, such as the transition from `B` to `B1`, is denoted by `T`, indicating a jump is taken; (2) an unconditional jump, like the transition from `A` to `D`, is represented by `J` to signify an unconditional jump instruction; (3) the sequential execution from one instruction to the next within the text segment is indicated by `NT` for jump *not* taken, as seen in the transition from `D` to `E`, or transitions without any jump instruction, such as from `B1` to `B2`.

5.1.2 Dynamic Control Flow

Like the objdump output, the *Control-Flow Graph (CFG)* provides a *static* representation of the control flow. It captures the relationships between different BBs and their transitions at compile time without actually executing the program. The *dynamic* control flow captures the concrete execution path of the program, representing the specific order in which basic blocks execute during runtime. It represents the specific order in which BBs are executed during runtime. Various factors influence the

Listing 5.1 – Objdump Snippet of Benchmark μQSI.
This objdump output snippet from μQSI displays one step of quicksort's sorting process. The code is partitioned into eight distinct BBs labeled `A` to `G`, each BB is highlighted in the code. BB `B1` diverges from the conventional pattern observed in other BBs by not concluding with a jump instruction; `B1` seamlessly transitions into the subsequent BB `B2`.

```
00100192 <qsort_iterative>:
;A
  100192: push %ebp
  100193: mov  %esp,%ebp
  100195: push %edi
  100196: push %esi
  100197: push %ebx
  100198: sub  $0x1c,%esp
  10019b: mov  0xc(%ebp),%ecx
  10019e: mov  0x10(%ebp),%edx
  1001a1: mov  %edx,%eax
  1001a3: sub  %ecx,%eax
  1001a5: lea  0x13(,%eax,4),%eax
  1001ac: and  $0xfffffff0,%eax
  1001af: sub  %eax,%esp
  1001b1: lea  0x3(%esp),%eax
  1001b5: shr  $0x2,%eax
  1001b8: lea  0x0(,%eax,4),%edi
  1001bf: mov  %edi,-0x20(%ebp)
  1001c2: mov  %ecx,0x0(,%eax,4)
  1001c9: mov  %edx,0x4(,%eax,4)
  1001d0: mov  $0x1,%edi
; --- jump to D - line 37
  1001d5: jmp  1001e4 <qsort_iterative+0x52>
;B1
  1001d7: mov  %edx,(%ebx)
;B2
  1001d9: add  $0x1,%eax
  1001dc: cmp  %esi,%eax
; --- conditional jump to F - line 57
  1001de: jl   10020f <qsort_iterative+0x7d>
;C
  1001e0: test %edi,%edi
; --- conditional jump to G - line 65
  1001e2: js   100220 <qsort_iterative+0x8e>
;D
  1001e4: mov  -0x20(%ebp),%eax
  1001e7: lea  (%eax,%edi,4),%ebx
  1001ea: mov  (%ebx),%esi
  1001ec: mov  -0x4(%ebx),%eax
  1001ef: sub  $0x4,%esp
  1001f2: push %esi
  1001f3: mov  %eax,-0x1c(%ebp)
  1001f6: push %eax
  1001f7: push 0x8(%ebp)
  1001fa: call 1000a4 <partition>
  1001ff: lea  -0x1(%eax),%edx
  100202: add  $0x10,%esp
  100205: cmp  -0x1c(%ebp),%edx
; --- conditional jump to B1 - line 26
  100208: jg   1001d7 <qsort_iterative+0x45>
;E
  10020a: sub  $0x2,%edi
; --- jump to B2 - line 28
  10020d: jmp  1001d9 <qsort_iterative+0x47>
;F
  10020f: mov  -0x20(%ebp),%ecx
  100212: lea  (%ecx,%edi,4),%edx
  100215: mov  %eax,0x4(%edx)
  100218: mov  %esi,0x8(%edx)
  10021b: add  $0x2,%edi
; --- conditional jump to D - line 37
  10021e: jns  1001e4 <qsort_iterative+0x52>
;G
  100220: lea  -0xc(%ebp),%esp
  100223: pop  %ebx
  100224: pop  %esi
  100225: pop  %edi
  100226: pop  %ebp
  100227: ret
```

dynamic control flow, such as runtime conditions, user input, external events, and other dynamic elements that determine the program's execution path in the CFG. It provides a more accurate representation of the program's behavior during its execution [Muc97]. In other words, the CFG is a *scheme* for all program's possible dynamic control flows, and a single dynamic flow is an *instantiation* of the CFG.

Regarding the exemplary BBs from Listing 5.1 and Figure 5.1, one possible complete dynamic control flow using *ten* static BBs is:

$$\mathtt{A} \rightarrow \mathtt{D} \rightarrow \mathtt{B1} \rightarrow \mathtt{B2} \rightarrow \mathtt{F} \rightarrow \mathtt{D} \rightarrow \mathtt{E} \rightarrow \mathtt{B2} \rightarrow \mathtt{C} \rightarrow \mathtt{G}$$

For this chapter, I distinguish between static and *dynamic BBs*. In contrast, a dynamic BB is defined similarly to its static counterparts: It starts at the jumped address and ends with an unconditional or taken conditional jump. A non-taken conditional jump or the consecutive execution of the next instruction in the text segment does not define a new dynamic BB. For the upper chain of BBs, this definition leads to a *squashed* version of the control flow with *seven* dynamic BBs in between each arrow:

$$\mathtt{A} \rightarrow \mathtt{D} \rightarrow (\mathtt{B1}, \mathtt{B2}) \rightarrow \mathtt{F} \rightarrow (\mathtt{D}, \mathtt{E}) \rightarrow (\mathtt{B2}, \mathtt{C}) \rightarrow \mathtt{G}$$

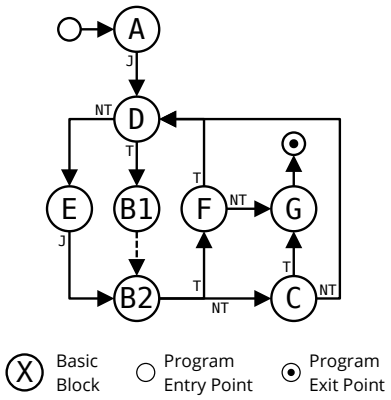


Figure 5.1 – Control Flow Graph for Listing 5.1.
The control flow graph represents the code from Listing 5.1. It visualizes the entry and exit points of the code, along with the individual BBs and their transitions. The `J` denotes unconditional jumps to another BB, and `T` label conditional jumps for taken and `NT` for not taken. The BBs `B1` and `B2` are in consecutive order in the text segment; thus the transition from `B1` to `B2` represents the program-flow execution without any jumps.

Therefore, dynamic BBs are always either equivalent to their static counterparts (e.g., `A`) or encompass a sequence of consecutive static BBs, when no jump occurs; the latter is the case when non-taken conditional jumps (e.g., (`D`, `E`)) or consecutive execution (e.g., (`B1`, `B2`)) occur. Consequently, the dynamic information regarding the jump behavior during the program flow defines the dynamic control flow and its dynamic BBs.

5.2 Fault-Space Region

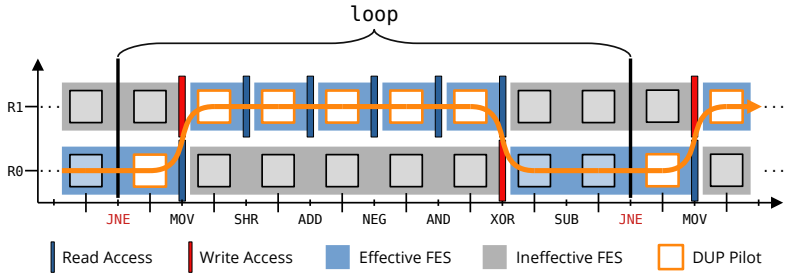
Despite the use of DUP (see Section 2.3.2.1 on page 47), an FI campaign can still become infeasible due to the large number of necessary pilots required. To address this issue, various heuristics (like the heuristics presented in Section 2.3.2.2 on page 48) exist to reduce further the number of necessary FIs. My developed *Fault-Space Regions (FSRs)* [▷Pus+19] also fall into this category. However, I base the method on the DUP technique, precisely its FESs, and deduce the required FIs using information obtained from the golden run, which incorporates the dynamic control flow of the program.

Figure 5.2 represent the basic idea behind the FSRs. Listing 5.2a shows a snippet of a CRC32 implementation pseudo code, and Figure 5.2b shows the corresponding FS. The highlighted points in the FS represent the pilots determined by DUP. The arrow in the figure represents the potential *data flow* of a flipped bit, passing from one loop iteration to the next, assuming no masking effects occur during the execution of instructions.

The fundamental concept of FSRs is based on examining FS segments defined by *region borders* (i.e., the `JNE` instruction in Figure 5.2) and selectively injecting only the FESs that cross these region borders. If a bit flip occurs within a region, it must eventually propagate across the region border to reach the system boundary. Thus, this propagation continues from region to region until an externally observable behavior is visible, akin to the concept of the fault propagation chain (refer to Section 2.1.3 on page 19). Thus, if any of the instructions between the borders of the FSR (e.g., `AND`) mask the bit flip, it will not cross any border anymore. Unlike the DFP from Chapter 4, FSRs do not analyze the concrete data flow of a bit but instead *approximate* the data flow; it considers FSRs as

```
1  loop:
2    R1 := MOV R0
3    R3 := SHR R3
4    R1 := ADD R1
5    R1 := NEG R1
6    R1 := AND R1, R4
7    R0 := XOR R1, R0
8    R2 := SUB R2
9    JNE loop
```

(a) Example Code



(b) Def/Use Pruning

Figure 5.2 – Potential Bit-Flip–Propagation Flow Through Loop Iterations.
Listing 5.2a shows the pseudo code for the inner loop of a CRC32 implementation. Figure 5.2b illustrates the corresponding FS segment, partitioned into FESs using DUP. The resulting pilots for complete coverage are highlighted. The arrow through the pilots indicates that any single bit flip occurring at one of these points of the FS will be propagated along this path. In this specific case, a bit flip from the previous loop iteration enters the current loop and continues to propagate in subsequent iterations.

black boxes through which data flows enter and exit, symbolized by the FESs crossing region borders. This reduces the FIs of the upcoming campaign to those FESs that would carry bit flips from one region to another.

Figure 5.3 illustrates the general concept of FSRs; you see the k-th FSR of the FS. This figure illustrates the three fundamental components of the FSR approach for reducing the number of pilots: (1) The FS is divided into FSR regions, which are segments of the temporal dimension, by applying a fixed rule to establish *region borders* (i.e., b_k and b_{k+1}). (2) The FESs of the effective FS, obtained through DUP, are categorized into two types: FESs that cross a region border are referred to as *outer FESs*, whereas those that do not cross any border are considered *inner FESs*. (3) Finally, the *pilots* and their corresponding weights from a defined *weight function* are determined based on the outer FESs and the *FSR weight*.

This section presents this generalized method that applies to an FI campaign independent of the FSs, architectures, programs, or ISAs. Later, I will present two possible instantiations of this generalized approach based on dynamic BBs and function scopes.

5.2.1 Determining Region Borders

The first step is to define the *region borders*. The determination of region borders is possible in various ways. Theoretically, the applier can use randomly distributed borders or uniformly distributed region borders with a fixed number of regions spanning the entire FS. However, during the development of this approach, it became evident that such non-deterministic and arbitrary determinations yield ineffective results.

To explore the potential of leveraging program properties, I further investigated BBs as the fundamental components of a control flow. By analyzing the golden run obtained from the FAIL* tracing tool and examining the objdump, I can extract the dynamic control flow as part of the program structure and use it for the deterministic calculation of FSRs.

Throughout the history of programming, programmers have intuitively divided programs into modular parts (regarding the *divide-et-impera principle* [Cor+09]), whether through jumps, recursion, or function calls. Compilers also divide program code into modular parts like BB, thus implicitly

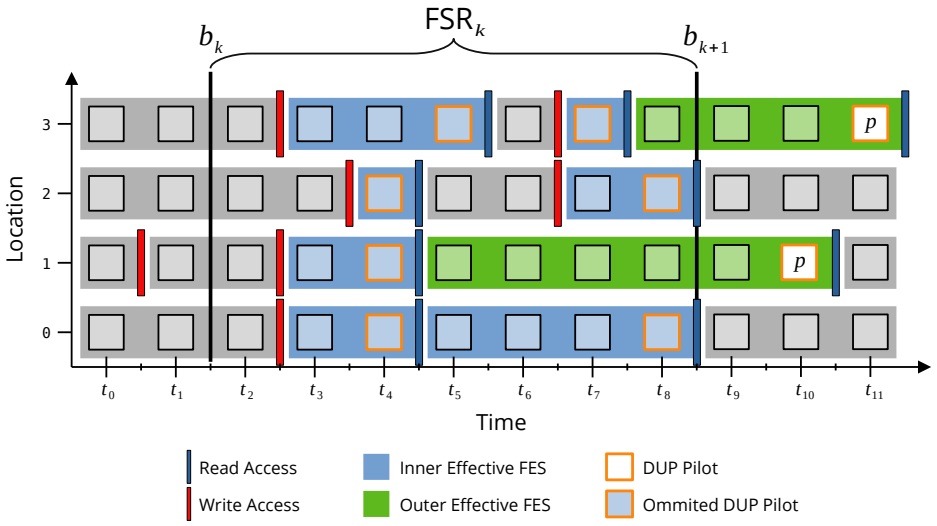


Figure 5.3 – Exemplary Illustration of a Fault-Space Region.
The effective FS and its corresponding FESs were determined using DUP in this exemplary FS. Based on this, the FS is divided by region borders. The border b_k defines the start of FSR_k, and b_{k+1} concludes it. For FSR_k, if an FES extends beyond the region border (i.e., it crosses the border b_{k+1}), it is called an *outer FES*. If an FES remains within the region, it is considered an *inner FES*. Applying FSRs determines a pilot (p) for every outer FES such as when applying DUP (i.e., inject-on-read). The sum of all effective FESs of FSR_k is the weight of this FSR. This FSR weight is distributed over the determined pilots of the outer FESs using a weight function. The rule for setting region borders and the weight function are freely definable. Omitting FIs into the inner FESs of an FSR reduces the number of pilots for this FS to only two (i.e., nine for DUP; see the highlighted points in this FS).

aggregating the system state at these points of the code. Therefore, leveraging the program's structure for analysis is natural and technically relevant.

In the context of this dissertation, I developed two region border rules that specifically address the modularity within a program: Basic-Block Regions and Function Call Regions. These rules allow a more fine-grained examination of the program structure, leading to practical analysis and optimization.

5.2.1.1 Basic-Block Region

The first rule developed for constructing FSRs is based on *dynamic BBs*, as presented in Section 5.1.2. With FAIL*, we obtain the objdump and a golden run during the tracing step.

The objdump provides static information such as the addresses of the instructions in the text segment and the instruction width. The instruction width refers to the memory space in the text segment or the address span until the next sequential and complete instruction in the text segment. This approach derives the sequence of instructions and their corresponding addresses by analyzing the golden run, which contains the sequence of instructions and their respective instruction widths from the objdump. This extracted information is one possible way to define the region borders within the FSR construction process. This approach provides a precise and complete representation of the program's dynamic control flow.

The execution of instructions in a computer system follows the principle of *sequentially* executing the instructions in the text segment, following the monotonically increasing instruction addresses generated by the IP register of the processor [PH21]. In the absence of jump instructions, each instruction's address plus its instruction width is the address of the consecutive instruction in the text segment. If this is not the case, a jump instruction has occurred within the dynamic program execution.

The code in Listing 5.2 exemplifies the principle that represents the developed rule. The variable `trace` represents the golden trace, a list containing instruction addresses executed in the program. The variable `objdump` encompasses various static information, including the `width` attribute, which is the instruction width. It iterates over the entire golden run, and if the calculated address `current_instr.address + objdump[current_instr.address].width` does not match the address of the consecutive instruction; it implies the execution of a jump instruction. These cases are recorded in the `region_borders` list, creating a set of timestamps representing each BB region's left border. This function determines dynamic BBs.

Listing 5.2 – Determination of Basic-Block–Region Borders.
The golden run or trace is executed step-by-step within the loop, processing each instruction sequentially. During execution, each instruction's current address is compared to the expected address of the next instruction in the text segment. If these addresses do not match, a jump has occurred in the program. To determine this, the address of the next instruction, represented by `next_instr.address` , is compared with the sum of the address of the current instruction and its instruction width. If a mismatch is detected, it is recorded in the variable `region_borders` . The resulting values in `region_borders` represent the borders for the FSRs, which corresponds, in this case, to the program's dynamic BBs.

```
1  def calc_borders(trace: list, objdump: dict) -> list:
2      region_borders = []
3      current_instr = trace.pop(0)
4      trace_time = 1
5
6      for next_instr in trace:
7          # check if a jump was executed
8          if current_instr.address + objdump[current_instr.address].width != next_instr.address:
9              # record the time as a region border
10             region_borders.append(trace_time)
11
12         trace_time = trace_time + 1
13         current_instr = next_instr
14
15     return region_borders
```

Although it is theoretically possible to examine each instruction and further divide the BBs based on not-taken conditional jump instructions, it would require checking for jumps in every individual (non-jump) instruction. Moreover, determining whether an instruction is a jump instruction heavily depends on the ISA and is not trivial due to the numerous jump instruction variants in different ISAs.[36] By using BB regions, I employ a generic approach using dynamic BBs that operate independently of the processor's ISA and avoid the need to check each instruction for jumps.

5.2.1.2 Function-Call Region

My second rule, a refinement of the dynamic BB approach, aims at defining more coarse-grained FSRs while considering function calls within the program. Each function call, indicated by the instruction `call`, followed by a jump back to the `call`'s origin using the `ret` instruction, represents

[36] For instance, the spectrum of various x86-ISA jump instructions consists `JE`, `JZ`, `JNE`, `JNZ`, `JB`, `JC`, `JNAE`, `JNB`, `JAE`, `JNC`, `JBE`, `JNA`, `JA`, `JNBE`, `JS`, `JNS`, `JP`, `JPE`, `JNP`, `JPO`, `JL`, `JNGE`, `JGE`, `JNL`, `JLE`, `JNG`, `JG`, `JNLE`, `JMP`, `JO`, `JNO`, `JCXZ`, `JECXZ`, and `JRCXZ` [Cor22].

an implicit unconditional jump. Therefore, `CALL` and `RET` and all its relevant variants[37] also serve as BB entry and exit points and are handled within the BB regions. A simple extension in the if condition in the code snippet from Listing 5.2 is required to create the FSRs at the function level: The code must explicitly check for `CALL` and `RET` instructions, which allows for identifying function boundaries and determining FSRs using function calls accordingly. This approach determines larger FSRs, where each FSR is a set of dynamic BBs.

5.2.2 Distinguishing Inner and Outer Fault-Equivalence Set

When applying DUP, the one-dimensional FESs are generated; each single one consists of a collection of FS points corresponding to a location l_j at different points in time. Each FES_j has a start $t_{\text{FES}_j,\,\text{start}} = \{\min t_i \mid \forall (t_i, l_j) \in \text{FES}_j\}$, representing the earliest occurrence, and an end $t_{\text{FES}_j,\,\text{end}} = \{\max t_i \mid \forall (t_i, l_j) \in \text{FES}_j\}$, representing the latest occurrence. Therefore, the temporal interval for an FES_j spans from time $t_{\text{FES}_j,\,\text{start}}$ to $t_{\text{FES}_j,\,\text{end}}$.

The method uses the identified region borders to determine the FESs extending beyond a region border. Each $b \in B$ in the set of region borders B represents a specific point in time within the FS, indicating the beginning of a new FSR. For instance, a first region border $b_0 = 5$ indicates that a jump instruction occurred between t_4 and t_5. By applying the following condition, we can quickly identify those FESs that correspond to the FSR_k by crossing the FSR closing border b_{k+1} (see Figure 5.3 for reference):

$$b_k \leq t_{\text{FES}_j,\,\text{start}} < b_{k+1} \leq t_{\text{FES}_j,\,\text{end}} \tag{5.1}$$

These FESs are considered as *outer FESs* and will be used to determine the pilots using FSRs. The remaining FESs are *inner FESs* and are not considered when determining the pilots.

5.2.3 Defining Fault-Space Region Pilots

Once the region borders of the FSRs and the original FESs of the DUP have been sorted based on whether the FESs cross a region border, the next step is to select pilots for the FI campaign. This step involves determining a *point* within the FS and assigning a corresponding *weight* to the pilot.

The points within the FSRs are chosen based on selecting the latest possible time, specifically, *inject on read* [Bar+05] for every *outer FES*. The choice is because any system failure visible outside of the system must have propagated through all FSRs until the end. For an activated fault involving an inner FES to be propagated and observed at the end, the activated fault must also manifest in an outer FES. In other words, the activated fault must traverse somehow through the outer FESs of the FSRs in order to reach the end of the program and be observed.

Although this *approximation* allows for saving FIs, it also leads to a certain degree of deviation, making applying FSRs a *non-precise* method. However, despite the deviation, applying FSRs reduces the required FIs significantly, as detailed in the upcoming evaluation section.

An accurate *weight function* is crucial to ensure *complete coverage* of the FS despite the deviations occurring by omitting the inner FESs. This weight function assigns appropriate weights to the selected pilots, considering the significance and impact of each FI in achieving complete FS coverage.

[37] Far fewer instruction variants typically need to be considered compared to jump instructions, making the computation manageable from this perspective. In the case of x86 architecture, only the `CALL` variants `CALL` and `SYSCALL`, as well as the return variants `RET`, `RETF`, `IRET`, and `SYSRET` exist [Cor22]. I have not considered system calls and interrupt variants for this dissertation as they have not been implemented in my benchmark portfolio.

5.2.3.1 Weight of a Fault-Space Region

The weights of the FSR pilots must consider both the inner and the outer FESs to ensure complete FS coverage.

The FS is divided into distinct temporal parts by defining the region borders. Each FES_j that starts within a region FSR_k, as defined by the borders, belongs to that specific FSR_k. Therefore, we can accurately define a single FSR_k as a set of FESs, which leads to the following expression for the definition of an FSR_k:

$$\text{FSR}_k = \left\{ \bigcup_j \text{FES}_j \;\middle|\; b_k \leq t_{\text{FES}_j,\,\text{start}} < b_{k+1} \right\}$$

The FSR pilots cover the set of all FESs in the FS, including the inner and outer FESs. As a result, these pilots cover an entire FSR and all potential FIs associated with it. The FSR's *weight* is determined to quantify the coverage provided by the pilots. This weight indicates the size of the absolute set of points in the FS covered by the FSR. The weight of the k-th FSR is calculated as the sum of the cardinalities of all FESs belonging to the FSR:

$$w_{\text{FSR}}(k) = \left\{ \sum_j |\text{FES}_j| \;\middle|\; \text{FES}_j \in \text{FSR}_k \right\}$$

The collective set of all FSRs corresponds to the set of all FESs after applying the DUP. Ultimately, this represents the complete effective FS, so considering all FSRs makes this method *complete* regarding the FS.

5.2.3.2 Evolved Weight Functions for the Fault-Space Region Pilots

Once the weights for all FSRs are determined, assigning these weights to the *pilots* within each FSR is necessary, which ensures that the total sum of weights for all pilots across all FSRs matches the weight of the effective FS. During my research, I developed two weight functions to address this requirement. Both weight functions use the *arithmetic mean* and *weighted arithmetic mean*, respectively.

Arithmetic Mean Weight Function

The first weight function w_{mean}, based on the arithmetic mean, distributes the weight w_{FSR} of the k-th FSR evenly among every pilot p_i within the FSR_k, which I define as follows, where P_{FSR_k} represents the set of pilots for FSR_k:

$$w_{\text{mean}}(p_i) = \left\{ \frac{w_{\text{FSR}}(k)}{|P_{\text{FSR}_k}|} \;\middle|\; p_i \in P_{\text{FSR}_k} \right\}$$

This weight function ensures a *uniform distribution* of an FSR weight among all pilots within FSR_k. It treats each pilot as equivalent necessary for propagating activated faults across FSRs, regardless of the individual sizes of the propagating outer FES used for an FI.

Weighted Arithmetic Mean Weight Function

The second approach distributing the weight of FSR_k among its pilots depends on the size of the corresponding outer FES. This weight assignment considers the individual influences of the pilots

within FSR_k, considering the sizes of their associated FES. The intention is to assign a higher weight to pilots associated with larger FES, inspired by the weighting strategy employed by DUPs.

Each outer FES within an FSR represents a portion of the FS directly and precisely covered by an FI, which means a single pilot covers the part covered by its associated FES. As the outer FESs potentially propagate activated faults from one FSR to another until the end of the program, it is reasonable to use the outer FESs and their weights to determine a suitable weighted distribution for the FSR_k pilots.

First, the part of the FSR containing only the FS points belonging to an outer FES is determined. Equation 5.1 defines the condition for identifying the outer FESs, which leads to the weight of the FSR with outer FESs only:

$$w_{\mathrm{FSR,\,outer}}(k) = \left\{ \sum_j |\mathrm{FES}_j| \;\middle|\; \mathrm{FES}_j \in \mathrm{FSR}_k,\; b_k \le t_{\mathrm{FES}_j,\,\mathrm{start}} < b_{k+1} \le t_{\mathrm{FES}_j,\,\mathrm{end}} \right\}$$

All FESs that do not satisfy this condition above are in the inner FESs part of the FSR. The summed sizes of all inner FESs added with the outer FSR weight result in the weight of the entire FSR w_{FSR}. Using the outer FSR weight $w_{\mathrm{FSR,\,outer}}$, the total weight of the FSR $w_{\mathrm{FSR}}(k)$ is distributed proportionally based on the sizes of the associated pilot's FES:

$$w_{\mathrm{wmean}}(p_i) = \left\{ w_{\mathrm{FSR}}(k) \cdot \frac{|\mathrm{FES}_j|}{w_{\mathrm{FSR,\,outer}}(k)} \;\middle|\; p_i \in \mathrm{FES}_j \right\}$$

So, the weight function w_{wmean} assigns more weight to pilots originating from larger FESs, as they precisely cover more points in the FS. Thus, w_{wmean} is thus more consistent with the weighting approach used in the DUP method.

5.2.4 Summary of Applying Fault-Space Regions

Applying FSRs aims to reduce the number of pilots required for *complete* FS coverage. The reduction is achieved by injecting a subset of P_{DUP} based on the DUP method. The following steps summarize my contributed method:

1. Develop a rule for setting the *region borders* (see Section 5.2.1). In my work, I have analyzed the program structure of the benchmarks and established rules based on dynamic BBs or function scopes. These FSRs are modular instruction sequences or temporally separated parts of the FS.

2. Use the region borders to determine the FESs of the DUP that *cross* a region border (see Section 5.2.2). Faults activated within an *inner FES* eventually manifest in one of the border-crossing *outer FES*. Therefore, the outer FESs represent the parts of the FS that propagate potentially activated faults across the FSRs.

3. Assign a *pilot* to each outer FES and determine pilot's weights using a *weight function* (see Section 5.2.3). The DUP pilots only of the outer FESs are FSR pilots as well and I have developed two weight functions for this purpose: One is an arithmetic mean, and the other is a weighted one.

As the inner FESs are only indirectly incorporated into the method through the weighting of the pilots in the outer FESs, applying FSR becomes an *approximation*. Although the approach provides *complete coverage* of the FS, it may add a *deviation* in the results. However, as the evaluation will

demonstrate, reducing the number of required pilots is significant *without* causing drastic deviations in correctness.

The resulting FSRs using the abovementioned rules above will be named *Basic-Block Region (BBR)* and *Call Region (CR)* for the rest of this dissertation.

5.3 Evaluation

In this section, I present the evaluation *results* of applying FSRs. Firstly, I discuss the two evaluation criteria: (1) the *reduced number of pilots* required to achieve complete FS coverage and (2) the *deviation* from the precise results obtained with DUP, considering that applying FSRs is an *approximation* method. Next, I present the findings regarding the *locality* of the results because I coarse the focus point of the FIs from individual instructions to more significant code regions. Finally, I briefly sketch the *validation* of the FSR method.

5.3.1 Fault-Space–Region Effectiveness and Result Deviation

The goal of applying the FSRs is to reduce the number of pilots required for complete coverage of the FS. To assess the *effectiveness* of this FI campaign acceleration method, I have defined the corresponding metric in Equation 3.1 on page 61. This metric quantifies the percentage reduction in the number of pilots needed for full FS coverage.

Since the pilots *only* pertain to the outer FESs of the FSRs and the inner FESs are indirectly accounted for through the weights of the outer FESs, applying FSRs is an *approximation* and may yield imprecise results. To evaluate the meaningfulness of these imprecise results, I consider the *deviation* of the results as a measure of accuracy.

DUP provides the ground truth to establish a baseline for comparison. The set of DUP pilots P_{DUP} corresponds to its results R_{DUP}. By examining R_{DUP} and identifying which FES each pilot belongs to, it is possible to make precise and comprehensive statements for each point in the FS. The union of all FESs represents the effective FS. The *classification-specific weight accumulation function* $w_{\text{acc}}(c)$ globally assesses the impact of a specific failure class c of the set of all occurring failure classes C and its share in the whole FS. The function w_{acc} using the single pilot weights $w(p_i)$ regarding its corresponding FESs is defined as follows:

$$w_{\text{acc}}(c) = \left\{ \sum_{i=0}^{|P_{\text{DUP}}|-1} w(p_i) \;\middle|\; p_i \in P_{\text{DUP}}, c \in C, (p_i, c) \in R_{\text{DUP}} \right\}$$

The function w_{acc} represents the process of traversing the entire FS point by point and recording each class that occurs. It captures the semantic equivalent of examining the FS comprehensively and accumulating the weights for each failure class encountered.

I also apply this weight function w_{acc} to the results obtained after applying FSRs for each failure class that occurs. If there were no deviation and the FIs yielded identical results compared to DUP, the accumulated weights for each failure class would be the same as for DUP. However, even a single different result leads to differences in the accumulated weights for the corresponding failure classes.

Therefore, the deviation δ resulting from applying FSRs, compared to the precise DUP, is measurable directly when comparing the individual accumulated weights for each failure classification.

These absolute differences in weight are then normalized using the *geometric mean*[38] $\overline{\chi_{\text{geo}}}$ over the total amount of points overall FESs or rather in the effective FS $\mathscr{E}$, allowing for cross-benchmark comparability. The deviation δ is thus defined as follows:

$$\delta = \overline{\chi_{\text{geo}}}\left(\left\{ \frac{|w_{\text{acc, DUP}}(c) - w_{\text{acc, FSR}}(c)|}{|\mathscr{E}|} \;\middle|\; \forall c \in C \right\}\right)$$

5.3.2 Results Overview

In this section, I present the evaluation results of applying FSRs. I begin by providing a comprehensive *overview* of the developed regions, including BBRs and CRs, and both weight functions I developed. Following this overview, I delve into the results for each benchmark, considering them as a *whole FS* and then separating them by *memory FS* and *register FS* for a more detailed perspective.

To enhance the understanding of the approximation itself and its characteristics, I will also present an analysis of the *locality of the results*. This analysis provides insights into how the application of FSRs affects the distribution and proximity of the FIs within the FS.

Finally, I sketch the *validation* of the evaluation, where I demonstrate the ability to produce identical (precise) results to the DUP with a specific configuration. This validation ensures the reliability of the FSR approach.

By analyzing and presenting these results, we gain valuable insights into the effectiveness and applicability of FSRs in reducing the number of required pilots and its impact on the precision of results.

First, I show initial findings across MiBench and Micro benchmarks from my benchmark portfolio to provide an initial impression in the preliminary section. In this context, I thoroughly examine the *whole FS*, encompassing its temporal dimension and all locations. This FS offers a comprehensive perspective on the entire system. It encompasses faults in all critical storage and computation components, including the processor, memory, and, indirectly, caches. Moreover, I analyze this both collectively and individually for registers and memory. Within each of these distinct FSs and using the geometric mean, I present the percentage reduction Δn in the required FI, along with the deviations δ from the results given by the DUP for the weight functions highlighted in Section 5.2.3.2. I provide these comparative insights for both BBRs and CRs.

5.3.2.1 MiBench Benchmarks

Table 5.1 shows an overview of all the MiBench benchmarks. When considering the *whole FS* under examination, FSRs achieve notable reductions in the required FI are achieved: −72.45 percent when applying BBRs and −87.51 percent when using CRs. However, applying the weight function w_{mean} and directly comparing the results achieved with R_{FSR} against R_{DUP} yields deviations of 18.66 percent for BBRs and 30.77 percent for CRs. By employing the weight function w_{wmean}, the deviations are considerably diminished compared to w_{mean}. The deviations are merely 1.13 percent for BBRs and 4.84 percent for CRs. This analysis highlights that, in general, the reductions in required FIs for complete FS coverage are higher when CRs are employed. However, deviations are also more significant in this scenario. From an overview perspective, the data demonstrates that BBRs showcase less deviation from actual results than CRs. This overview also shows that using w_{wmean} leads to considerably lesser result deviations than using w_{mean}.

[38] The geometric mean $\overline{\chi_{\text{geo}}}$ is a statistical measure commonly used to calculate a set of values' central tendency or average. It is computed by taking the n-th root of the product of the values x_i, where n is the number of values in the set: $\overline{\chi_{\text{geo}}} = \sqrt[n]{x_0 \cdot x_1 \cdot \ldots \cdot x_{n-1}}$. The geometric mean is particularly useful when dealing with datasets that have multiplicative relationships between the values [LR22].

Fault Space			Whole	Memory	Register
Δn	[%]	BBR	−72.45	−11.64	−83.72
	[%]	CR	−87.51	−39.90	−97.30
δ mean	[%]	BBR	18.66	20.95	25.74
	[%]	CR	30.77	22.66	58.73
δ w.mean	[%]	BBR	1.13	0.10	20.94
	[%]	CR	4.84	3.58	45.09

Table 5.1 – Evaluation Overview of all MiBench Benchmarks.
The table illustrates the reduced percentage of necessary FIs, denoted as Δn, and the deviations δ given as the geometric mean for the MiBench Benchmarks. The shown deviations pertain using the two presented weighting functions, w_{mean} and w_{wmean}. The column *whole FS* refers to the entire FS, defined as $\mathscr{F} = T \times L$. This FS is divided into two distinct segments: the *memory FS* and the *register FS*. This separation allows a more profound view of the underlying evaluation. This data is further distinguished using the FSR rules BBR and CR.

Directing our attention to the second column, which focuses on the separate analysis of the FS containing only *registers*, we observe an even more substantial reduction. However, this reduction leads to notably higher deviations in the results. CRs achieve a remarkable −97.3 percent reduction. Nevertheless, this reduction comes with deviations of 58.73 percent and 45.09 percent, indicating significant disparity in the outcomes. On the other hand, when employing BBRs, although the reduction in FIs is slightly less, the deviations are comparably lesser than CRs.

Shifting our focus to the *memory FS* analyzed independently, the reductions achieved are more modest. However, these smaller reductions lead to correspondingly minor deviations in the results. Specifically, in the case of the BBRs within the MiBench benchmarks and with the w_{wmean} weight function, there is an −11.64 percent reduction in the number of FIs with an impressively minimal deviation of only 0.1 percent.

I will delve into this point in more detail in the subsequent sections. However, I want to sketch this already: The reduction observed in the context of registers can be attributed to the generally smaller FESs because data in registers do not last long compared to data in the memory. Consequently, more FESs in the register FS are categorized as *inner FESs* when applying the FSR approximation, leading to their exclusion. Although this results in substantial reductions, it also yields suboptimal precision in the outcomes. Conversely, the memory FS behaves distinctly differently. The FESs are comparatively larger, resulting in a considerable portion of *outer* memory's FESs, and hence not excluded. Consequently, in contrast to the register FS, a smaller proportion of the inner FESs' weight needs to be distributed to the outer FESs in the memory context which results in a more feasible deviation of the results.

5.3.2.2 Micro Benchmarks

In Table 5.2, I present an overview of the Micro benchmarks. In general, there is a notable reduction in the number of FIs, yet the overall deviation is notably more pronounced when compared to the MiBench benchmarks. However, there is an exception to this trend found within the memory FS, mainly when BBRs and w_{wmean} are employed, resulting in a minimal deviation of only 0.74 percent. When scrutinizing the deviations of individual benchmarks within this category, the results vary, ranging from 0.004 percent (μFIB) to 1.95 percent (μMIX) when using BBRs and w_{wmean}. However, the deviations to the MiBench benchmarks are substantially higher under any other configuration.

Fault Space			Whole	Memory	Register
Δn	[%]	BBR	−61.00	−10.29	−70.81
	[%]	CR	−76.89	−29.45	−86.84
δ mean	[%]	BBR	46.94	18.65	29.93
	[%]	CR	50.37	19.17	59.90
δ w.mean	[%]	BBR	11.23	0.74	18.97
	[%]	CR	32.90	6.92	59.51

Table 5.2 – Evaluation Overview of all Micro Benchmarks.
This table shows the same type of data like as Table 5.1 but overall Micro benchmarks instead.

The FSR's poor performance in the Micro benchmarks depends on their minimal workload. These benchmarks consist of only 54 to 2 093 instructions (the Mibench benchmarks have at least 40 647 instructions), resulting in a small number of BBRs ranging from 14 (μLSUM and μMIX) to 490 (μFIB) or a small number of CRs ranging from 5 (μLSUM and μMIX) to 183 (μFIB). The limited workload and the unfavorable ratio between the limited number of FSRs and the number of instructions leads to the classification of many FESs as inner FESs, resulting in notably high deviations in the obtained results. Due to this, the Micro benchmarks are unsuitable for evaluating the FSR in terms of instruction volume.

5.3.2.3 Overview Summary

At this point, I present a summary so far and highlight the key points we will delve into next:

Impact of Fault Spaces: When considering the FS as a *whole* or *segregating* it into memory and register locations, *different outcomes* emerge regarding the reduction of FIs and the deviation of the results.

- Application of the FSRs method proves effective across the whole FS, resulting in notable reductions and manageable deviations.
- In the register FS, reductions are substantially higher, but deviations are also more pronounced due to the small size of FESs, causing the omission of many inner FESs.
- Reductions within the memory FS are comparatively lower, yet deviations are significantly reduced compared to the register FS, primarily due to the larger FESs and increased presence of outer FESs.

Effect of Weighting Functions: The two developed weighting functions yield strongly varying results.

- The relatively simplistic weight function, w_{mean}, leads to substantial deviations and appears unsuitable for the FSR method.
- In contrast, the weight function w_{wmean} produces more accurate results and seems suitable for applying FSRs.

Comparison of FSR Types: The two developed FSR rules exhibit distinct behavior.

- BBRs, with their finer granularity, yield more precise results than CRs, accompanied by legitimate reductions in FIs.
- Using CRs leads to more inner FESs, leading to higher reductions in FIs and elevated outcome deviations.

Workload and Program Structure Impact: The workload and program structure plays a pivotal role in determining the effectiveness of the FSR method.

- Micro benchmarks, characterized by low workloads and a limited number of generated FSRs, are not well-suited for this method.
- In contrast, the MiBench benchmarks, featuring significantly higher workloads, prove more suitable for FSR application, resulting in statistically more stable results concerning deviation.

In the following sections, I will delve into more comprehensive results for each benchmark, reinforcing the initial observations. To streamline the amount and presentation of the displayed data, I will entirely exclude unsuitable aspects. Therefore, I will refrain from discussing the w_{mean} weighting function or the Micro benchmarks from this point onward.

Despite the apparent unsuitability of the FSR method for the register FS, I present detailed results for comprehensiveness. This dissertation thoroughly addresses these results as an integral part of the complete FS.

5.3.3 Fault-Space–Separated Results

In the subsequent sections, I will present more detailed evaluation results. Initially, I will present the findings for the whole FS, followed by the distinctions between the memory-only and register-only FS to explore the effectiveness of the FSRs on different location types.

5.3.3.1 Whole Fault Space

In this investigation, I thoroughly examine the whole FS of all MiBench benchmarks, encompassing the complete temporal dimension and all accessible system locations. These locations encompass all registers and memory addresses available within the ISA layer.

In Table 5.3, I begin by presenting the foundational data for each benchmark. This first dataset is the workload, defined as the number of instructions within the benchmark, effectively representing the length of the temporal dimension T of the FS. Additionally, the table provides the total size of the $\mathscr{F}$ and the size of the effective FS $\mathscr{E}$.

As a reminder, the effective FS constitutes the portion of the FS that the DUP effectively considers and shapes into individual FESs. For each of these FESs, a pilot, essentially an FI, is determined within the DUP process. The total quantity of these FIs is aggregated and presented as n_{DUP}. Each n_{DUP} of each benchmark is the *ground truth* (see Section 3.2.1 on page 59) for every benchmark evaluation regarding the number of required FIs. Importantly, all these data across all benchmarks are within the same order of magnitude, ensuring their comparability.

Lastly, I added the number of FSRs (#regions) determined using the BBR or CR rule. All this data is derived from the program trace, ensuring it is the same across all detailed examinations, namely register FS and memory FS, except the FS sizes $|\mathscr{F}|$, which may vary.

MiBench- Whole FS			miBC	miBFD	miBFE	miQS	miRDD	miRDE	miSHA
$\|T\|$			54 434	55 647	54 951	45 774	70 824	70 450	40 647
$\|\mathscr{F}\|$		$[\cdot 10^9]$	0.08	2.17	2.16	1.78	3.85	3.81	0.34
$\|\mathscr{E}\|$		$[\cdot 10^9]$	0.07	1.89	1.88	1.62	3.51	3.46	0.24
#regions	BBR		10 666	4 768	4 824	6 211	3 905	3 933	5 251
	CR		1 431	409	411	985	390	454	42
n_{DUP}		$[\cdot 10^6]$	2.52	3.50	3.46	2.93	4.17	4.17	2.67
Δn	BBR	[%]	−70.39	−74.61	−74.90	−64.42	−76.73	−77.51	−72.14
	CR	[%]	−92.22	−85.01	−84.88	−85.75	−81.77	−81.87	−97.95
δ w.mean	BBR	[%]	1.56	0.94	0.77	0.47	2.09	1.93	0.84
	CR	[%]	3.61	6.02	6.26	1.72	5.61	2.93	9.10

Table 5.3 – Detailed Results of the Whole Fault Space for each MiBench Benchmark.
This figure presents the results of applying FSRs to the MiBench benchmarks. It shows the temporal dimension's length of the FS $|T|$, the overall size of the FS $|\mathscr{F}|$, and the effective part of the FS $|\mathscr{E}|$. Additionally, it lists the total number of regions (#regions) using BBRs or CRs. For comparison, the table shows the count of DUP pilots (n_{DUP}) and the reduction achieved in pilots using BBRs or CRs (Δn). Lastly, the table denotes the deviation observed in the determined FI experiment outcomes compared to the results of the precise DUP's FI experiments (δ).

Table 5.3 shows the results for the whole FS of each MiBench benchmark. When using BBRs, the outcomes exhibit a maximum deviation of 2.09 percent (miRDD), whereas the reductions in the required FIs range from −64.42 percent (miQS) to −76.73 percent (miRDD).

Conversely, the average deviation increases to 4.84 percent when employing the coarser CR rule. However, the FI reduction remains substantial, starting from −81.77 (miRDD) percent and increasing (excluding miSHA) to −92.22 (miBC) percent. Notably, miSHA poses challenges due to its limited number of function calls, resulting in a higher deviation, as indicated by 9.1.

BBRs are suitable for the entire FS, whereas CRs offer a viable option when a slightly higher deviation is acceptable.

5.3.3.2 Memory Fault Space

In Table 5.4, I present detailed results for the FSR method applied to the memory FS of each MiBench benchmark.

For miBC, miRDE, and miSHA, BBRs cause maximal deviations, reaching 0.78, 0.35, and 0.2 percent, respectively. These deviations occur alongside reductions in FIs ranging from −22.9 to −35.86 percent. However, deviations caused by BBRs in other cases are negligible.

Furthermore, the FI reduction (Δn) achieved by BBRs ranges from approximately 1 percent (miBFD and miBFE) to −35.86 percent (miRDE). The minor reduction observed for the blowfish benchmarks (miBFD and miBFE) is due to significant and frequently occurring FESs that span region borders. In cases where memory is not used for intermediate results, the BBR rule converges toward the DUP method.

For CRs, the table displays the most substantial deviation, approximately 8.56 percent, for the miSHA benchmark, coupled with a significant FI reduction of −90.38 percent. This outcome results from limited function calls, leading to only 5 251 FSRs. It highlights that CRs are less practical for programs with few function calls.

MiBench- Memory FS			miBC	miBFD	miBFE	miQS	miRDD	miRDE	miSHA
$\lvert\mathscr{F}_{\text{mem}}\rvert$		$[\cdot 10^9]$	0.06	2.16	2.14	1.76	3.83	3.79	0.33
$\lvert\mathscr{E}_{\text{mem}}\rvert$		$[\cdot 10^9]$	0.06	1.88	1.87	1.62	3.49	3.44	0.23
n_{DUP}		$[\cdot 10^6]$	0.12	0.62	0.62	0.57	1.17	1.16	0.52
Δn	BBR	[%]	−22.90	−1.27	−1.18	−13.95	−34.69	−35.86	−23.78
	CR	[%]	−45.11	−23.14	−23.34	−40.03	−39.65	−38.96	−90.38
δ w.mean	BBR	[%]	0.779	0.003	0.004	0.076	0.183	0.353	0.201
	CR	[%]	4.113	4.182	4.073	1.099	4.945	1.646	8.557

Table 5.4 – Detailed Results of the Memory Fault Space for each MiBench Benchmark.
This table shows the same type of data as in Table 5.3 but focuses only on the locations of the memory FS. The values for the $|T|$ and #regions are equivalent to those in Table 5.3.

Apart from miSHA, CRs reduce the number of FIs by at least −23.14 percent and up to −45.11 percent, with deviations ranging from 4.18 percent to 1.65 percent.

In summary, CRs (excluding miSHA) and BBRs significantly reduce memory FS with moderate approximation errors.

5.3.3.3 Register Fault Space

For the register FS, the results *present* a distinct perspective, as shown in Table 5.5. The table reveals that the register FS is two orders of magnitude smaller than the memory FS and is notably denser with short-read FESs, which is evident from the minor FS reduction achieved by the DUP method.

This discrepancy comes from the limited number of registers compared to the multitude of touched memory cells and the brief retention time of register data. Across both rules, the FSR method can save a minimum of −72.78 percent of all FIs (miBC), albeit at the cost of significantly higher deviations. Deviations can reach up to 42.59 percent for BBRs (miRDE) and 102.74 percent for CR (miSHA).

MiBench- Register FS			miBC	miBFD	miBFE	miQS	miRDD	miRDE	miSHA
$\lvert\mathscr{F}_{\text{reg}}\rvert$		$[\cdot 10^7]$	1.57	1.60	1.58	1.32	2.04	2.03	1.17
$\lvert\mathscr{E}_{\text{reg}}\rvert$		$[\cdot 10^7]$	1.23	1.19	1.13	0.83	1.43	1.31	0.78
n_{DUP}		$[\cdot 10^6]$	2.40	2.88	2.84	2.36	3.00	3.01	2.15
Δn	BBR	[%]	−72.78	−90.31	−90.90	−76.66	−93.20	−93.63	−83.75
	CR	[%]	−94.58	−98.25	−98.24	−96.83	−98.28	−98.48	−99.76
δ w.mean	BBR	[%]	12.68	14.55	8.72	15.64	57.53	67.11	26.39
	CR	[%]	36.53	42.73	54.47	17.09	44.15	42.59	102.74

Table 5.5 – Detailed Results of the Register Fault Space for each MiBench Benchmark.
This table shows the same type of data as in Table 5.3 but focuses only on the locations of the register FS. The values for the $|T|$ and #regions are equivalent to those in Table 5.3.

These substantial deviations stem from a prevalent register-usage pattern that conflicts with the assumption that the FSR method captures region computation results by selecting the border-crossing FESs. In situations where intermediate results are computed in registers but subsequently written to

memory before leaving the region and the scoped FS (for instance, caller-saved registers for CR), the relevant data flows occur within the FSRs, bypassing the FSR method in memory.

As a result, FSRs are unsuitable for a register-only FS when intermediate results traverse through main memory.

5.3.4 Locality of the Results

For a resilience assessment of systems or programs, the focus often extends beyond the end-to-end error rate (see Section 2.1.4 on page 21). Comparing different program parts or sections becomes pivotal. For instance, detailed FI results empower system designers to decide which functions require additional software-based hardening measures, such as triple-modular redundancy.

The application of FSR represents a *shift* in focus from individual instructions to larger code regions. Consequently, the method can no longer distinguish between adjacent instructions but instead calculates local error rates for each FSR, which remains feasible because the FSR method approximates, independently for each region, from the FI results of the outer FESs to all FESs within the region.

Furthermore, since FSRs align with the program structure and partition the execution trace along the time axis, they still enable differentiation between various program phases and code blocks. For instance, with BBRs, the method can assess whether the initial execution of a loop body is more vulnerable than subsequent iterations.

In the ensuing sections, I will explore how well the FSR method with the BBR rule can provide localized results and whether this *locality* is sensitive to specific benchmarks, FSs, or failure classes. As CRs exhibited higher deviation at the whole program level, I will concentrate solely on BBRs in this section.

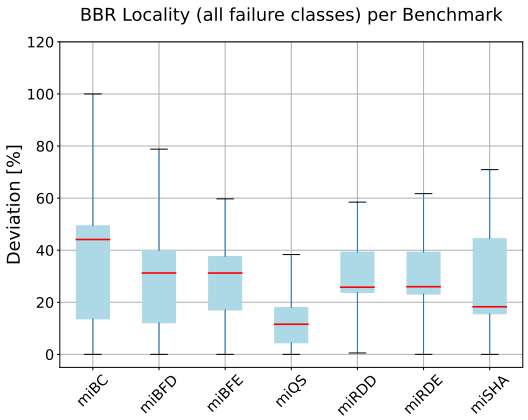
For each BBR, I compute two error-rate vectors: a precise one and an approximated one. These vectors encompass results for each of the four failure classes. For the precise baseline, I weight the FI results for inner and outer FESs according to their FES size. In contrast, for the approximated vector, I apply weights to the results of the outer FESs according to w_{wmean}. Subsequently, I calculate relative deviations element by element and employ the geometric mean to summarize the BBR-specific precision.

5.3.4.1 Benchmark- and Fault-Space Sensitivity

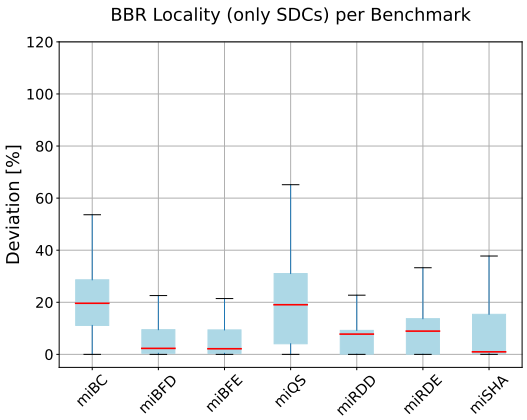
To begin with, I aim to assess the sensitivity of the local results concerning different benchmarks and the three used FSs. As a general expectation, higher precision at the whole program level (Table 5.3-5.5) should correspond to better local results.

For the register FS, which already exhibited challenges at the whole-program view, the method demonstrates significant deviations when scrutinizing individual BBRs, which results in median deviations ranging from 29.6 percent (miQS) to the maximum 100 percent (miRDD and miRDE). Even five 75-percent quantiles show up to 100 percent deviation from the actual result. Once again, this highlights that the FSR method is less suitable for a register-only FS, as local results can effectively *bypass* the region borders in memory.

Figure 5.4athe boxplot depicting BBR-result deviations considering the combined FS. We observe that the median deviations remain below 45 percent, with a 75-percent quantile extending up to 49.4 percent (miBC). Consequently, good results in the whole-program approximation (Table 5.3) correspond to more precise results for the local error-rate function. For instance, miQS displays a median deviation of 11.4 percent.



(a) Boxplots representing all benchmarks across all failure classes. Each data point represents the geometric mean of relative deviations within the local error-rate vector for a single Basic-Block Region.



(b) Boxplots showcasing all benchmarks for the Silent Data Corruption failure class. Each data point signifies the relative deviation within the local SDC error rate for an individual BBRs.

Figure 5.4 – Deviations in Local Results for the Whole Fault Space per Benchmark.
The red line denotes the median, the boxes enclose the 0.25 and 0.75 quantiles, and the whiskers extend to the last value within 1.5 times the interquartile distance. For clarity, the box plots do not show the outliers.

Moving to the memory FS, I observed the most precise local results, as anticipated from the whole-program approximation. In this case, all benchmarks exhibited a median relative statistical error of zero, and only one benchmark displayed a 75-percent quantile unequal to zero: miQS remained below 0.1 percent deviation in the 0.75-quantile. Consequently, BBRs are well-suited for the whole-program approximation and provide precise local results for the memory FS.

5.3.4.2 Failure-Class Sensitivity

In addition to benchmark and fault-space sensitivity, I have also delved into the sensitivity of BBR-specific results across different failure classes. This investigation is particularly relevant because, in practice, some failure classes, especially SDCs, hold more significance in assessing the resilience of a program segment or code block. Therefore, I have conducted a detailed examination of local deviation results for the combined FS, which provides the most comprehensive coverage of program behavior.

Figure 5.5 shows boxplots depicting relative error-rate deviations across all benchmarks, grouped by their failure class. Notably, traps show the highest degree of imprecision (median: $\approx$ 63 percent), which heightened imprecision suggests that such outcomes more frequently result from FIs within a BBR.

Conversely, the BBR rule yields the most favorable results for the most critical failure class: SDC. Here, the median deviation is 11.1 percent, and the 75-percent quantile remains below 20 percent.

With this insight, Figure 5.4b concentrates exclusively on SDCs and presents the per-benchmark quality of BBR-local results. This figure uses the same boxplot as Figure 5.4a but with an SDC class filter applied.

Observations reveal that median precision for local SDC rates is generally higher for all benchmarks except for miQS. Median values span between 0.97 percent for miSHA and 19.62 percent for miBC and 0.75 quantiles range from 9.22 percent (miRDD) to 31.04 percent (miQS). Since local

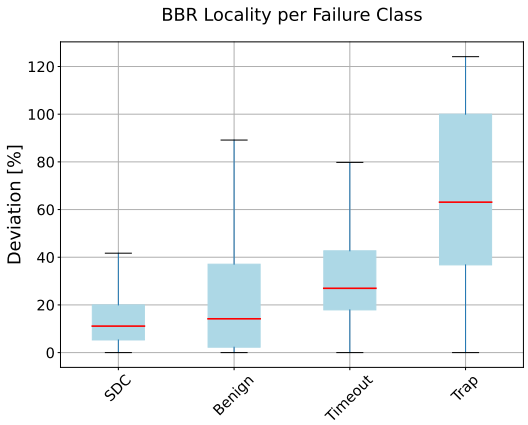


Figure 5.5 – Deviations in Local Results for the Whole Fault Space per Failure Class.
These boxplots illustrate data points from all benchmarks across various failure classes for the whole FS and represent the median, quantiles, and whisker values as in Figure 5.4.

results for the SDC class are often more precise than those of other failure classes, the data flows resulting in SDC incidents frequently cross a BBR border and are effectively captured by the FSR method.

This investigation into local error rates underscores that the BBR rule can provide a high degree of locality. Particularly for SDC failures, which are the most critical, the FSR method with the BBR rule yields low deviations from precise FIs that encompass the entire FS but at significantly lower FI costs.

5.3.5 Validation

To validate the FSR approach, I devised a process to calculate the *equivalent* set of DUP pilots. The sets of FSR pilots and DUP pilots are equivalent when the FSR method considers all FESs as outer FESs, meaning that only FESs crossing region borders are taken for pilot determination.

To achieve this, I established a trivial region border rule in which a new region border is set after each instruction. As a result, the FSR method creates a total of $|T|$ FSRs, each with a temporal length of 1.

For an FES to be an outer FES, it must *cross* a region border. FESs with sizes of two or more satisfy this condition. However, FESs with sizes of 1 do not cross any border and, by definition, remain classified as inner FESs at this stage of the validation process.

Instructions that consecutively read the same bit generate FESs with a size of 1. In such instruction sequences, DUP can also not optimize the number of required pilots based on the read and write accesses, resulting in a pilot with a weight of 1.

The corresponding pilots are equivalent when comparing FESs of size two and larger between DUP and FSR. The FESs of the remaining pilots of DUP, which have the size 1, are equivalent to the inner FESs when the FSR approach is applied. Therefore, if one pilot is assigned to each of these inner FESs, these pilots are equivalent to the DUP pilots with a weight of 1. Finally, the pilots' weights and, thus, the sum of all pilot weights are the same in both cases; therefore, equivalent FI campaigns occur.

5.4 Summary of Program-Structure–Guided Approximation of Fault Spaces

Fault Injection is a powerful means for quantifying a program's resilience against transient hardware faults, an essential aspect of assessing the functional safety of critical systems. However, the number of required FIs escalates *rapidly*. Even with precise reduction methods like the Def/Use Pruning, achieving complete and precise *Fault Space (FS)* coverage becomes *impractical*.

To address this challenge, I introduce the *Fault-Space Region (FSR)* approximation method. FSRs significantly reduce the number of necessary FIs while maintaining relatively low deviation in failure classification. I construct these FSRs from *program structures*, such as basic blocks and call instructions, by analyzing the program trace objdump. Faults are only injected into data flows that *cross* a region border (as explained in Section 5.2.1). A single pilot is defined and injected for these *outer Fault-Equivalence Set (FES)*, whereas the inner FESs are omitted from injection (as presented in Section 5.2.3 and Section 5.2.2). To distribute the weight of the omitted FESs to ensure FS full coverage, I have developed two weight functions: w_{mean} and w_{wmean} (as detailed in Section 5.2.3.2).

I compare the required faults between the precise DUP and my FSR method in the evaluation. I also quantify the approximation error between the two methods when applied to three different FSs: general-purpose registers, main memory, and both concurrently.

The MiBench benchmarks provided valuable insights into the *effectiveness* of the FSR method. For seven MiBench benchmarks from the automotive and security categories, the FSR method achieved an average reduction in the required faults for the entire FS by 77.51 percent, with a relative approximation error of less than 2.09 percent regarding the whole FS. However, the workload must not be too small for high effectiveness. Therefore, the micro benchmarks in the dissertation's benchmark portfolio were unsuitable.

The evaluation revealed that the FSR method for the register FS faces challenges due to the short length of FESs. Consequently, fewer FESs cross region borders or cause faults to bypass into memory. Additionally, the weight function w_{mean} is unsuitable, leading to significant deviations. In contrast, w_{wmean} performs much better and is the preferred weight function. Comparatively, the *Call Region (CR)* rule demonstrated higher reductions in FIs but resulted in higher deviations than the *Basic-Block Region (BBR)* rule.

In light of these findings, the optimal approach for applying the FSR method is to consider the whole FS or the memory FS, using the BBR rule and the w_{wmean} weight function. This generally appliable combination strikes a balance between reducing FIs and minimizing deviations.

Furthermore, I present the *locality of the results* obtained through the FSR method. For this, I compare the precise error-rate vector (DUP) with the error-rate vector approximated by the FSR method. The FSR method demonstrate a solid ability to maintain the locality of results, particularly concerning the failure class Silent Data Corruption, with a median deviation of 11.1 percent.

TRY IT OUT: Program-Structure–Guided Approximation of Fault Spaces in FAIL*

For those interested in the source code of the evaluated Fault-Space Region instances, based on basic blocks and function scopes, it can be found within the *prune-trace tools* directory of the FAIL* GitHub repository:

`https://github.com/danceos/fail`

6

Timeout-Detection Methods for Fault-Injection–Experiment Acceleration

> You build on failure. Use it as a stepping stone and close the door on the past. Don't try to forget the mistakes, but don't dwell on it.
>
> JOHN R. "JOHNNY" CASH (1932–2003)

The complete duration of an FI campaign comprises n essential pilots. These pilots encompass the entire FS, and each of these single FI experiments requires time duration of t_{exp}. Consequently, the total duration of the FI campaign is $t_{\text{cpn}} = n \cdot t_{\text{exp}}$. In the previous two chapters (4 and 5), I have introduced two methods to reduce the overall campaign duration t_{cpn} by minimizing the number of pilots n.

This chapter delves into the FI experiment time t_{exp}, a factor that has not yet received attention in my dissertation. Herein, I provide a detailed examination of the *timeout* failure class, signifying instances where program execution fails to terminate after an FI. I will describe the existing approaches for handling timeouts in general and in the context of FAIL* and explore the potential gains from optimizing the handling of timeouts, or rather the *prediction* of timeouts. This problem presents a formidable challenge as it can be reduced to the renowned *Halting Problem* [Gö31; Tur37], thus defying deterministic solutions.

Furthermore, this chapter outlines various approximation methods I have previously developed and evaluated. These methods are introduced to address the timeout prediction challenge in greater depth. Notably, the method that has demonstrated the best performance is *ACTOR*. I provide a

comprehensive exposition of this effective timeout prediction method within the context of FI campaigns at the ISA layer. This method can significantly reduce the total campaign duration t_{cpn}, and I present detailed evaluation results to support this claim.

My colleague presented ACTOR at the SAFECOMP conference [▷Tho+22] in 2022, and I was the paper's third author. The development of ACTOR occurred in a master's thesis supervised by me. I shared my prior research experiences in the timeout context, contributed to the conceptualization of ACTOR, designed the benchmarks used, analyzed evaluation results for further improvement, and contributed to paper writing.

6.1 Fault-Injection Experiment Time

In this section, I delve into the *temporal* dimension of individual FI experiments and establish some definitions to aid in comprehending this chapter. The duration of an FI experiment plays a pivotal role in shaping the overall runtime of an FI campaign, given that a campaign encompasses a multitude of such experiments. For instance, as evidenced in the collaborative research [▷Tho+22], a total of $2.8 \cdot 10^7$ injections determined by DUP results in the execution of $1.3 \cdot 10^{12}$ instructions post-injection, which entails a *sequential* runtime of approximately 13 CPU days when employing FAIL* on the evaluation setup, operating at a simulation rate of 1.16 MHz.

Even within FI frameworks like FAIL*, which exhibit high parallelizability due to their client-server architecture (see Section 3.1 on page 53), it remains advantageous to minimize the average duration of individual FI experiments in order to decrease the entire FI-campaign runtime further or to reduce resources used by the FI framework. To evaluate the *potential* for reducing the runtime of FI experiments, I will expound upon several key aspects: the relationship between the number of FIs, the count of simulated instructions, and the distribution of failure classes throughout the campaign's runtime.

This *workload analysis* of the benchmarks used in this chapter offers insights into the distributions of failure classes and the number of executed instructions. Whereas other approaches, like those presented in Section 2.3.3 on page 49, concentrate on the failure class SDC, my focus in this chapter is on examining the failure class of *Timeout (TO)*.

6.1.1 Experiment Time Line

First and foremost, let me clarify what I mean by the concept of FI experiment *time*. To aid in this understanding, Figure 6.1 illustrates the time frame of a FI experiment along with some relevant intervals.

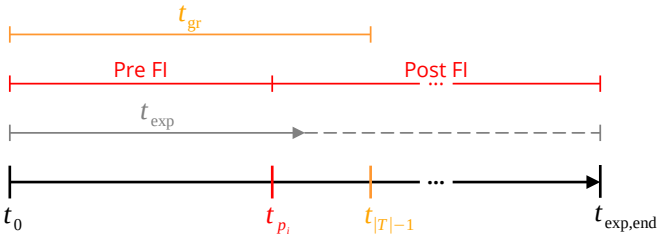


Figure 6.1 – Fault-Injection Experiment Time Line.
This figure shows various definitions for time points and intervals used in this chapter. An FI experiment starts at time t_0 and concludes at time $t_{\text{exp, end}}$; thus, its duration is t_{exp}. The interval of the *golden run* is the period t_{gr} for potential FIs and is defined by t_0 and $t_{|T|-1}$. Each pilot p_i corresponds to a specified FI time t_{p_i} between t_0 and $t_{|T|-1}$. The interval before the t_{p_i} is the *pre-FI* interval, and the one after is the *post-FI* interval. $t_{\text{exp, end}}$ can occur before $t_{|T|-1}$ depending on the behavior of the SUT and the FI (e.g., the SUT triggers a trap immediately after the FI).

An FI experiment begins at time t_0, signifying the initial state of the SUT (i.e., before the execution of the first instruction). The associated pilot, denoted as p_i, specifies the injection time point t_{p_i}. This FI time point is determined regarding the golden run and aligns with one of the recorded time points in the set T, which falls within the interval between t_0 and $t_{|T|-1}$. The temporal length of the golden run is t_{gr}. The time interval preceding t_{p_i} is the *pre-FI time*, whereas the period after t_{p_i} is the *post-FI time*.

Following the injection, the experiment progresses until it reaches the time point $t_{\text{exp, end}}$. At the moment of $t_{\text{exp, end}}$, when the SUT *terminates*, the FI framework determines a failure class (see Section 3.1.2 on page 56) for this specific FI. In cases where the SUT *does not terminate* (detected by a time threshold being reached for example), the FI framework records a *timeout* for the pilot. The time point $t_{\text{exp, end}}$ can occur *before* $t_{|T|-1}$, such as when the SUT triggers a trap immediately after the FI, but it is after t_{p_i} as per definition.

The elapsed time from the starting point at t_0 to $t_{\text{exp, end}}$ constitutes the duration of the FI experiment, denoted as t_{exp}, which is the focus of this chapter. Depending on the circumstances, this duration t_{exp} may not only be shorter but can also extend beyond the original duration of the golden run. Such an extension could occur if, for instance, an FI affects a loop's counter variable, resulting in its value remaining valid but decreased. Consequently, the loop would run for a more extended period compared to the golden run. This illustrates how the FI can alter the control and data flow after the FI occurs.

Furthermore, as discussed in Section 3.2.1.3 on page 62, t_{exp} is the sum of both the program's runtime t_{prg} of the SUT and any associated cumulated overhead t_{ov}. This overhead can arise from various sources, including the simulation of the SUT and the FI framework, or the implementation of acceleration methods.

When injecting a specific fault, FI frameworks like FAIL* expedite the program's execution to the injection time t_{p_i} and then execute the FI. This acceleration mainly applies to the pre-FI time. Depending on the specific FI framework, this forwarding process can be optimized through techniques like checkpointing [KDC05; Ber+02] or hardware-assisted breakpoints [SRS14], as detailed in Section 2.3.3 on page 49.

However, the post-FI execution cannot be accelerated to the same extent because the FI-induced control flow may deviate from the golden run and behave *unpredictable*.

6.1.2 Fault-Injection Campaign Workload

First and foremost, I provide an overview of the overall workload associated with the benchmarks in this dissertation. Using the precise DUP pilots ensures complete coverage of the FS, as detailed in Section 2.3.2.1 on page 47. Each of these pilots leads to a corresponding FI, resulting in an FI outcome categorized within one of the predefined failure classes, as discussed in Section 3.1.2 on page 56.

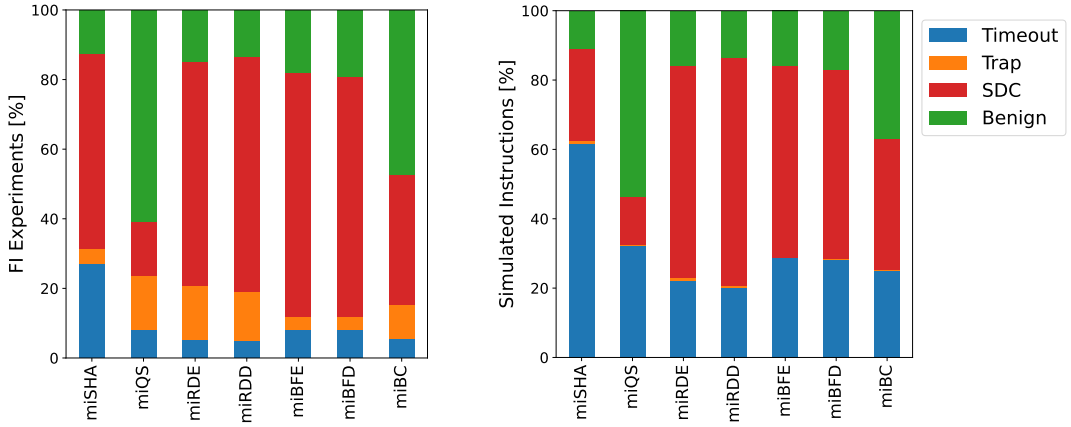
I will delve into the distribution of experiments within an FI campaign and how it correlates with the number of instructions (i.e., the concrete workload). By the end of this section, I will conclude and pinpoint the specific aspect within this context that will be the primary focus of this chapter.

6.1.2.1 Number of Fault-Injection Experiments and its Simulated Instructions

Figure 6.2a presents the distribution of FI results as percentages for all MiBench benchmarks. The predominant failure class across all benchmarks is SDC. SDCs account for a percentage ranging from 15.6 (miQS) to 70 (miBFE) percent in the individual distributions and 49.1 percent (geometric mean) overall MiBench benchmarks. This indicates that when additional data independent of the control flow is injected, the SUT can continue functioning correctly but will eventually terminate with a divergent result. The next most significant class overall is *benign*, with variations from 12.6 (miSHA) to 60.7 (miQS) percent and an overall portfolio percentage of 22 percent. In these FIs, the injection proves ineffective, ultimately leading to the correct termination of the SUT due to the masking effects of individual instructions or the regenerative algorithm semantics. The failure classes *trap* and *timeout* follow closely and exhibit similar proportions. Traps range from 3.7 (miBFE) to

15.8 (miRDE) percent, with 8 percent overall, while timeouts vary from 5.1 (miRDE) to 27 (miSHA) percent, with 7.9 percent overall.

Based on this data, Figure 6.2b shows the number of simulated instructions *after* the occurrence of an FI until the end of the benchmark. The count of instructions executed by experiments ending in a trap is negligible. Compared to the proportion of FI experiments that result in an SDC, the proportion of simulated instructions drops significantly overall. The proportions of injections associated with an SDC experiment vary from 13.7 (miQS) to 66 percent (miRDD). Across all benchmarks, the proportion of instructions relative to the failure class proportions decreases to 40.2 percent. The number of simulated instructions for benign FI experiments ranges from 10.9 (miSHA) to 53.7 percent (miQS), with a slight decrease overall across all MiBench benchmarks, reaching 19.9 percent. Notably, the instructions for a timeout experiment have increased compared to all other failure classes and the FI experiment count from Figure 6.2a. The proportions range from 20.1 (miRDD) to 61.7 (miSHA) percent, accounting for 29.2 percent across all MiBench benchmarks.



(a) Executed FI Experiments

(b) Simulated Instructions Post-FI

Figure 6.2 – Distributions of Experiment Results and Instructions After Fault Injection. Figure 6.2a displays the distribution of resulting failure classes across all MiBench benchmarks in percent. In contrast, Figure 6.2b shows the number of simulated instructions executed *post-FI* and how they correspond to the resulting failure classes.

6.1.2.2 Instructions over Fault-Injection Campaign Time

As depicted in the overview provided in Figure 6.2, SDCs constitute the majority of simulated instructions. However, compared to the proportion of the FI experiment amount, the proportion of simulated instructions has notably increased during timeouts, which *suggests* that after an FI, a significant portion of instructions are executed in FI campaigns corresponding, to the relatively low number of FI experiments that conclude with a timeout.

Figure 6.3 illustrates the progression of an FI campaign using the miQS benchmark as an example. The x-axis of this stacked histogram represents the period of the most extended FI experiment, specifically experiments where the FI initiates before executing the first instruction. On the y-axis, the figure shows the count of ongoing experiments within the FI campaign at various points *post-FI*. The different colors further categorize these counts, representing emerging failure classes. The initial, almost proportional increase in the number of ongoing experiments is due to the delayed timing of

single FIs during the experiments, which varies based on the pilot's definition and occurs at different time points. Additionally, the figure highlights the end time of the golden run $t_{|T|-1}$ for reference.

Figure 6.3 – Temporal Distribution of Experiments Segmented by Failure Classes for the Benchmark MiBench Quicksort.
This figure illustrates the histogram of the number of FI experiments at different time points *post-FI* for the benchmark miQS. The potential time frame for FIs, as defined by DUP pilots, spans from the first to the last instruction of the golden run. Each experiment runs for a maximum of three times the duration of the golden run, which is the timeout threshold in this case. The number of experiments at each time point is divided into failure classes. The integrals in the histogram correspond to the number of executed instructions overall, and the segment integrals based on the failure classes are equivalent to the boxplot data in Figure 6.2b.

The integrals of this histogram for each failure class provide the count of executed instructions post-FI, corresponding to the data already presented in Figure 6.2b. Therefore, the increased proportion of timeout instructions mentioned earlier is also evident in this case.

As demonstrated in Figure 6.2a, most of the experiments fall into the benign failure class. A significant portion of these experiments terminated after the regular end time of the golden run but before reaching 1.1 times the runtime of the golden run t_{gr}. Experiments that lead to an SDC also show a similar trend.

At any specific point, the total number of ongoing experiments represents the hypothetical scenario where all experiments run simultaneously. For instance, at instruction number 20000, FAIL* executes approximately one million experiments, and of those, 9.4 percent continue to run until reaching the timeout threshold. Beyond the histogram's fall (about 49000 instructions), most experiments conclude and result benign or SDC. Consequently, the proportion of experiments resulting in timeouts increases. When most experiments terminate, about 49000 instructions, slightly less than 1.1 times the golden-run duration, approximately 70.6 percent fall into the timeout category. This proportion steadily rises until the experiment reaches the threshold; the timeout threshold is *three times* the runtime of the golden run (ca. 137000 instructions).

Determining an appropriate TO threshold is challenging, making it an open research question. The conventional approach, irrespective of the FI context, involves extending the program runtime by

a specific *factor* and selecting this extended time as a TO threshold. This factor is inherently *arbitrary*; in the literature, factors between two and five are chosen without substantial justification [SB19; Kal+15; MV12], and Leo and colleagues even suggest a factor of ten [DL+12]. However, when considering the trade-off between FI campaign runtime and the potential correctness of the ground truth, I chose a factor of three throughout my work, which seems to be a suitable threshold value regarding my benchmarks and one commonly used value in the research community.

6.1.2.3 Conclusions

Since the integral in the histogram of Figure 6.3 represents the number of executed instructions post-FI, a reduction directly equates to *optimizing* the overall runtime of an FI campaign. As outlined in Section 2.3.3 on page 49, numerous studies have tackled predicting the outcome of an FI experiment, specifically whether an experiment results in an SDC or not. In contrast, timeout research is relatively scarce within simulation-based FI at the ISA level. Consequently, my work has centered on optimizing the post-FI time of FI experiments that conclude in a timeout.

The potential for optimization is readily apparent in Figure 6.3. If it were *feasible* to accurately predict a timeout immediately after an FI, the ongoing experiment would result in a timeout, and the portion of experiments ending in timeouts in Figure 6.3 would vanish entirely. In the case of miQS, this would result in significant savings in executed instructions, particularly after the threshold of roughly 1.1 times the golden-run runtime.

However, predicting timeouts is reducible to the *Halting Problem*, so timeout prediction is not deterministically solved. Although it is not solvable in the generic case, with enough additional information and constraints, there are *potential opportunities* for handling TOs. Using a TO threshold provides a practical way to approximate the existence of a timeout.

Nevertheless, selecting an appropriate threshold is nontrivial. Higher thresholds increase the probability of correctly classifying an FI experiment as a timeout, but they also *expand* the area represented in Figure 6.3, thus raising the overall FI-campaign runtime. Conversely, lowering the threshold to save overall FI-campaign runtime may lead to *misclassifications*. For instance, if a threshold of 70000 instructions exists in the case of miQS, 8.5 percent of the remaining experiments are misclassified as timeouts, although they eventually terminate as non-timeouts.

As the research on timeout prediction in the context of ISA-layer FI is limited in the FI community, I have focused on optimizing the FI experiment time t_{exp} by using acceleration methods that predict timeout classifications. This, in turn, reduces the overall FI-campaign runtime t_{cpn}.

6.2 Timeouts in Fault-Injection Campaigns

This section will delve into handling TOs, explicitly focusing on *TO detectors*. I will describe the properties and approaches for implementing TO detectors. Following that, I will explore how the decisions made by these detectors *impact* FI campaigns as a whole, considering the interplay between correct and incorrect TO determination. Finally, I will assess the tangible potential for FI-campaign runtime savings that arise from effective TO predictions.

6.2.1 Common Timeout Handling

In defining a *TO detector*, we need to specify two fundamental properties: *when* the detector triggers and the *decision logic* it employs.

6.2.1.1 Timeout Thresholds

The timing of the detector's decision, often referred to as the *TO threshold*, can be set in two ways within the context of FIs. As illustrated in Figure 6.4, the two possibilities are as follows:

Relative Threshold $t_{\text{thr,rel}}$ Here, the TO threshold is set relative to the injection time, t_{p_i}. The implementer defines a period x_{rel}, and the detector triggers at $t_{\text{thr,rel}} = t_{p_i} + x_{\text{rel}}$.

Absolute Threshold $t_{\text{thr,abs}}$ In this case, the TO threshold is an absolute time, which selects an absolute, fixed period or relating it to the program runtime, t_{prg}, or the golden-run runtime, t_{gr}. For instance, setting an absolute threshold of three times the golden-run duration is also feasible: $t_{\text{thr,abs}} = 3 \cdot t_{\text{gr}}$.

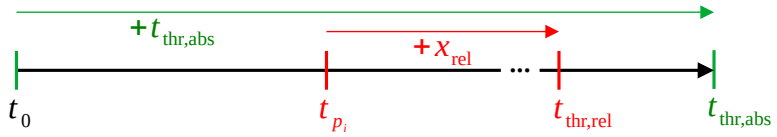


Figure 6.4 – Two Possibillities for Defining Timeout-Detection Thresholds.
This figure illustrates two possibilities for establishing a TO threshold on a timeline: (1) The first approach involves a defined period x_{rel} relative to the injection time t_{p_i}, resulting in a relative threshold $t_{\text{thr,rel}} = t_{p_i} + x_{\text{rel}}$. (2) Alternatively, the threshold can be defined with an absolute value $t_{\text{thr,abs}}$.

Relative thresholds are suitable when no golden run is available or when the SUT's execution time lacks determinism and predictability. They are also helpful when a consistent and homogeneous TO behavior across campaigns is desired.

On the other hand, absolute thresholds, independent of any specific runtimes, are universally applicable. However, using fixed arbitrary values for absolute thresholds can lead to unnecessary extensions of the FI-campaign runtime. Therefore, it is recommended to establish a reference to the program or SUT, such as setting the threshold as a multiple of the golden run duration (e.g., $3t_{\text{gr}}$) [SB19; Kal+15; MV12; DL+12]. Absolute thresholds relating to the program runtime provide a more contextually relevant TO determination.

Both approaches offer flexibility in tailoring the detector's behavior to the specific requirements of an FI campaign.

6.2.1.2 Detector Decision Logics

There are two approaches to how a detector reaches its decision:

Static Detection In this approach, the detector is triggered at the predefined threshold and decides *independently* of the system's current state. Once the execution reaches the threshold, it results in a TO.

Dynamic Detection In contrast, when detecting TOs dynamically, the detector evaluates the system's state and employs specific *heuristics* to determine whether a TO has occurred. This approach allows for flexibility; the experiment can conclude with a TO, the SUT terminates, or the SUT continues if the TO decision is negative.

These approaches depend on the FI campaign's requirements and the SUT's characteristics.

Static detection is simple but may lead to premature TOs, whereas dynamic detection can be more accurate but requires careful design and implementation of heuristics. Static detection is a simple and deterministic way to handle TOs, but it may not be the most efficient or accurate method. It is suitable for situations where there is consistent TO behavior across FI campaigns or when there is no golden run to reference. In addition, *hard deadlines* applied in real-time systems provide natural static thresholds.

Dynamic detection is more sophisticated and adaptable; it can respond to system behavior variations and may avoid unnecessary TO recordings if it functions correctly. It is useful when the system's execution time varies, is periodic, has unknown aspects, or a more fine-grained approach to TOs is desired.

Furthermore, static thresholds have no overhead, as they do not process data but only make a static decision, whereas dynamic detectors and decision-making add overhead overall.

6.2.1.3 Summary

To construct a TO detector within the FI context, the developer must make crucial decisions based on the properties previously outlined. However, as discussed earlier, *all* combinations of the two properties are valid in the context of FI. In the case of FAIL*, a *relative* and *static* TO detector has been implemented, allowing the threshold to be specified in absolute milliseconds.

As per my literature review, within the simulation-based–FI context, static detectors seem to be the norm. These detectors are typically triggered based on either an absolute or relative threshold. Developing effective *dynamic* TO detectors in this domain remains an ongoing research question.

There is a subtle but significant distinction between the two properties concerning the *effectiveness* (Section 3.2.1 on page 59) of the TO detector as an FI-campaign acceleration method: (1) Adjusting the TO detector threshold, regardless of its dependence on the SUT's program runtime, primarily impacts the overall *runtime* of the FI campaign. (2) In contrast, the choice between static and dynamic TO detector decision logic can significantly influence the *accuracy* of determining whether a TO has occurred. Incorrect decisions of the TO detectors are possible, which can affect the *quality* of the outcome.

Regardless of implementing a TO detector there should always be a *framework threshold* with a *static* decision where the SUT automatically terminates, an essential and standard TO mechanic. Using FAIL*, this framework threshold for the static determination of a TO is an *absolute* value and does not refer relatively to the pilot's injection time.

6.2.2 Quality of Timeout Detectors

Implementing a TO detector as an FI-campaign acceleration method introduces another dimension to its overall effectiveness: the *quality* of TO detector decisions. In this section, I delve into the interplay between optimized FI-campaign runtime and the enhanced quality of TO detector decisions. As the sole focus on runtime optimization is no longer adequate in this context, I also examine how this quality can be *quantified*, primarily through *statistical binary classification*.

6.2.2.1 Campaign Runtime and Timeout-Detection–Quality Trade-Off

Decisions from TO detectors are prone to *misclassifications*, as already sketched above. However, when evaluating the effectiveness of TO detectors as FI-campaign acceleration methods, focusing solely on the saved overall FI-campaign runtime is *insufficient*.

Looking at the two extreme cases in choosing the FI threshold, it becomes evident that FI campaign runtime savings are not the only factor to consider: (1) In one case, a static detector is triggered immediately at the start of an FI experiment and classifies the experiment as a TO. This approach is highly effective in reducing the FI campaign's total runtime. However, it results in classifying *all* FI experiments as TOs, regardless of their actual outcome, significantly impacting the quality of the static TO decisions. (2) On the other extreme, if we assume an infinite threshold as feasible, this implies that the Halting Problem, in essence, is solvable. In this scenario, *all* detector decisions would be correct, leading to the highest possible quality for the TO detector. However, no meaningful statement occurs from the runtime optimization perspective.

These extreme cases demonstrate the trade-off between FI-campaign runtime optimization and *static* detector quality:

The higher the TO detection threshold during an FI experiment,
the higher the quality of the TO detector decision.

A higher average runtime of FI experiments, achieved by setting a high threshold, naturally leads to a longer overall FI campaign runtime, which is indeed poor. However, the implementer must consider the *trade-off* between shorter FI-campaign runtime versus higher TO detector quality during the TO detector design process.

The theorem mentioned above is also applicable to dynamic detectors. However, dynamic detectors differ from static detectors because they can continue the FI experiment after making a correct or wrong decision and repeat the decision when the corresponding used data or system state changes. Nonetheless, the quality of a dynamic detector still relies on the timing of the decision, that is, the detector threshold. Indeed, it also depends on the decision logic that considers the system state of the SUT, providing opportunities to exert control over the program flow.

6.2.2.2 Statistical Binary Classification

Assessing campaign runtime savings when using a TO detector alone is insufficient; the correctness of the decisions is also essential, as they can potentially result in misclassifications. I employ *statistical binary classification* to measure TO detection's decision correctness (i.e., the quality).

Binary classification categorizes elements of a set into *two* distinct groups, or classes, using a predefined classification rule [Bis06]. In my case of TO detectors as FI campaign acceleration methods, the classification rule is the logic of the TO detector, and the result is the detector's prediction of whether an FI experiment results in a TO or not.

Fundamental combinations can occur based on the *correct* FI experiment classification and the TO detector decision assigned by the implemented detector logic to evaluate the quality of a TO detector. As mentioned earlier, there is no inherently correct set of classifications in the context of TO detection. Employing a static TO detector with an *extremely* high threshold suffices for evaluation. During my preliminary investigations, I selected a framework threshold ranging between 100 s and 200 s, while the runtime of a MiBench benchmark program is well below one second. The factor of > 100 times the program runtime for this framework threshold may be impractical for real-world applications but serves its purpose for evaluation and establishing a (good enough) *ground truth* within the challenging nature of the Halting Problem.

Classifications

Figure 6.5 illustrates four combinations between the *actual* and *predicted* classifications [Bis06].

The terms *positive* or *negative* describe the TO detector's decisions. If the TO detector determines that the FI experiment ends in a TO, the decision is *positive*; otherwise, it is *negative*. Depending on

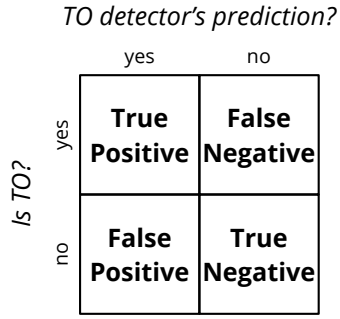


Figure 6.5 – Binary-Classification Combinations for Timeout-Detection Evaluation.
This table displays the four possible combinations for binary classification in evaluating *Timeout (TO)* detectors. One dimension stands for the decision made by the TO detector, where *Positive* indicates a TO prediction, and *Negative* represents a non-TO prediction. The other dimension represents a failure classification from the FI experiment's ground truth, whether a TO (*True*) or not (*False*). This table allows us to derive the correctness of the decisions and the type of misclassification (either FP or FN).

the correctness of these decisions, four combinations exist in the context of TO detection during an FI experiment:

True Positive (TP) The FI experiment without TO detectors proceeds until it reaches the framework threshold and is classified as a TO. The TO detector correctly predicted the TO at its threshold, allowing the SUT to terminate at the detector's threshold. As long as the TO detector threshold is before the framework threshold, it saves FI experiment runtime, and the prediction is correct.

True Negative (TN) The FI experiment without TO detectors is usually completed and classified as non-TO. The TO detector correctly predicted that the FI experiment would *not* result in a TO and the FI experiment proceeds. Finally, the SUT terminates as it would without an implemented TO detector; thus no runtime is saved in this case, but the detector's decision was correct.

False Negative (FN) The FI experiment without TO detectors proceeds up to the framework threshold and is initially classified as a TO. The TO detector, triggered at its threshold, *incorrectly* decides that there is *no* TO; this is a wrong prediction. The FI experiment continues until it reaches the framework threshold, at which point it is correctly classified as a TO. In this case, the TO detector's decision was wrong, but it has no ultimate consequences for the *correctness* of the FI campaign's resulting failure classifications. This does not save FI-experiment runtime, but the time between the detector and framework thresholds could have been saved if the prediction had been accurate. Thus, a detector should produce as few FNs as possible because every FN wastes potential runtime savings, but FNs do not distort the detector's correctness.

False Positive (FP) The FI experiment without TO detectors is usually completed and classified as non-TO. The detector *incorrectly* decides that the FI experiment will end in a TO. This decision is incorrect and the FI experiment terminates earlier than an FI experiment without a detector. In this case, time is saved, but the outcome is wrong. Unlike FNs, FPs distort the TO detector's accuracy and the FI campaign's *correctness*. A detector should not produce any FPs because any FP is an actual prediction error and leads to incorrect campaign results.

Quantifying the Timeout Detector's Quality

Now that I have provided a more detailed explanation of the binary classifications and their meaning, the next step is to *quantify* the quality of the TO detector across a whole FI campaign with various *rates* [Bis06]. With n planned pilots, n FI experiments start, and, as described in the previous section, there will be a total of n classifications concerning the TO detector's decision,.

The first rate is the *TP rate*, also known as *sensitivity* or *recall* [Bis06]. This rate represents the proportion of correct TO detections compared to the sum of correct detections (TP) and the number of wrong non-TO classifications (FN). Analogously, there is the *TN rate*, also known as the *specificity* [Bis06], which signifies the proportion of all correctly detected non-TO FI experiments compared to the number of correct non-TO detections (TN) and wrong TO detections (FP). The definitions for these rates are the Equation 6.1 below, where, for example, #TP stands for the number of TO detector decisions that correctly detected a TO during the execution of the planned FI campaign:

$$\text{TP rate} \coloneqq \frac{\#\text{TP}}{\#\text{TP} + \#\text{FN}} \qquad \text{TN rate} \coloneqq \frac{\#\text{TN}}{\#\text{TN} + \#\text{FP}} \tag{6.1}$$

If both these rates equal 1, the TO detector functions perfectly and does not make any misclassifications.

The complementary rates to TP and TN are the *FN rate* and *FP rate*,[39] which are defined as follows [Bis06]:

$$\text{FN rate} \coloneqq \frac{\#\text{FN}}{\#\text{TP} + \#\text{FN}} \qquad \text{FP rate} \coloneqq \frac{\#\text{FP}}{\#\text{TN} + \#\text{FP}} \tag{6.2}$$

As described in the previous section, an FN does *not* impact the correctness of failure classifications in the FI campaign. At an FN rate of 1, the FI campaign runtime reverts to what it would be if each FI experiment had reached the *framework threshold*. The FN rate does not affect the correctness of the FI campaign but serves as an indicator of potential FI campaign runtime savings the lower it is. An FN rate of 0 suggests that potential FI campaign runtime has been saved but does not guarantee the correctness of individual FI experiment classifications.

The FP rate is higher when more FI experiments are wrongly classified as TO compared to the number of correctly detected non-TO (TN) and wrongly detected TOs (FP). Accordingly, the FP rate reflects the correctness of the FI experiment classifications by a TO detector because every FP results in a wrong failure class within the FI campaign.

As the FN rate indicates, the rates are insufficient *individually* and must always be considered together. Since complementary rates exist, I will not delve further into the FN and TN rates; instead, I will consider the TP and FP rates. Thus, the TP and FP rates *together* quantify the quality of the TO detector decisions. An optimal and ideally deciding TO detector has a TP rate of 1 and an FP rate of 0.

6.2.3 Potential Fault-Injection–Campaigns Runtime Savings

This section delves into the *theoretical* runtime savings achievable through TO detectors. To quantify an FI campaign runtime saving potential, I define the *ground truth* for this chapter and outline the potential FI campaign runtime savings of an *optimal* TO detector (i.e., the upper bound). Following this, I analyze the implications of FI campaign runtime savings and how TP and FP rates influence these outcomes.

[39] The FN rate is complementary to the TP rate and the FP rate is complimentary to the TN rate.

6.2.3.1 Ground Truth in this Dissertation

A sufficiently high framework threshold is necessary to determine a ground truth in the TO detection context. After defining the framework threshold, the FI framework proceeds with the campaign as planned. The resulting failure classes are used to assess the TO detector's performance.

As mentioned earlier, I initially employed an exceedingly high, albeit arbitrarily chosen framework threshold, ranging from 100 s to 200 s, in scenarios with an average program runtime of less than one second. I assume a high threshold make TOs highly probable to be correctly determined. Although framework thresholds exceeding 100 times the golden-run runtime t_{gr} suffice to evaluate TO detector correctness, they are not practical for real-world applications or the productive utilization of FI frameworks.

Executing FI campaigns with a static $3t_{\text{gr}}$ framework threshold in FAIL* establishes the ground truth in the TO context. This ground truth serves a dual purpose: it defines both the baseline FI-campaign *runtime* and the *failure classes* used for comparing the performance of the implemented TO detectors under evaluation.

In this dissertation, I will use the term $3t_{gr}$ *detector* to reference the process in FAIL* of recording a TO at the selected absolute framework threshold $3t_{\text{gr}}$.

For the evaluation, this chapter focuses explicitly on the MiBench benchmarks only. Through my research for this chapter, I discovered that the runtimes of the Micro benchmarks need to be longer to obtain statistically stable and significant evaluation results in evaluating implemented TO detectors.

6.2.3.2 Optimal Timeout Detector

The ground truth defines one extremum of the FI-campaign runtime savings.

The other extremum is a detector that *dynamically* decides ideally at the injection time t_{p_i} whether the running FI experiment ends in a TO. I define the (hypothetical) *optimal* TO detector OPT, which stops all timeouts at the pilot's FI time t_{p_i} and thus reaches the theoretical optimum. As the optimal decision-maker, OPT acts with a TP rate of 1 and an FP rate of 0. Therefore, the application of OPT *removes* all post-FI instructions, shown in the stacked boxplot in Figure 6.2b or the TO integral in Figure 6.3.

Table 6.1 summarizes the percentage of post-FI instructions in the ground truth of the MiBench benchmarks, which corresponds to the TO percentages in the stacked boxplots of Figure 6.2b. If OPT were implementable, the percentages of these post-FI instructions listed in Table 6.1 would be saved, ranging from 20.05 to 61.67 percent, depending on the benchmark, representing the upper bounds of the potential savings regarding post-FI instructions.

MiBench Benchmark		miBC	miBFD	miBFE	miQS	miRDD	miRDE	miSHA
TO Post-FI Instr.	[%]	25.18	28.36	28.73	32.26	20.05	22.21	61.67

Table 6.1 – Percentages of Post-FI Instructions in Timeout Experiments.
This table illustrates the overall percentage of post-FI instructions executed from FI experiments within the ground truth, concluding with a TO for each MiBench FI campaign. The values correspond to the TO proportions detailed in Figure 6.2b.

The design space of possible dynamic detectors is between these extremes (OPT and $3t_{\text{gr}}$). Hence, each implemented dynamic detector adds to the framework-based $3t_{\text{gr}}$ detector, keeping FI experiments *bound*.

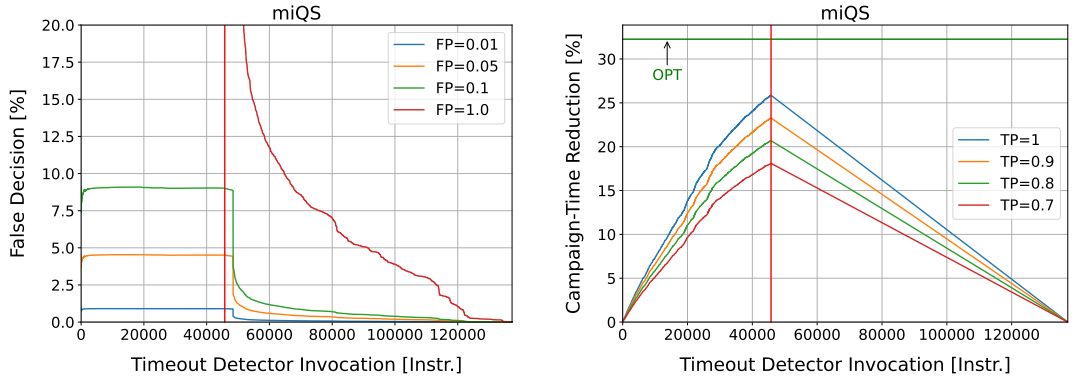
6.2.3.3 Analysis

Real-world detectors often have a non-zero FP rate. When applied to a population with many non-TO experiments, this can result in many FPs. These FPs can *skew* the failure class and are considered more significant than FNs, which mainly *only* extend the FI campaign duration.

To illustrate this, Figure 6.6a demonstrates the impact of the FP rate for an absolute detector on the percentage of false decisions for the MiBench benchmark miQS; the vertical line represents the time t_{gr}. Prior to t_{gr}, a detector that is 90 percent correct makes incorrect decisions in roughly 9 percent of cases. However, after t_{gr}, even a detector that labels all experiments as TOs (FP=1) becomes practical. This effect is even better for detectors with lower FP rates.

Consequently, detectors should operate after t_{gr}, which also rules out relative detectors as they could often trigger *before* t_{gr}.

Conversely, there is a compelling argument to activate the detector as early as possible to maximize its effectiveness and prevent the execution of stuck FI experiments. In this context, Figure 6.6b provides insight into the reduction in FI-campaign runtime achieved by absolute detectors with varying TP rates compared to the $3t_{gr}$ detector.



(a) False Decision Percentages of Hypothetical TO Detectors with Different FP rates

(b) Campaign-Runtime Savings of Hypothetical TO Detectors with Different TP rates

Figure 6.6 – Influence of Timeout-Detector Quality on the Campaign's Correctness and Runtime Savings.
These figures show the influence of different quality characteristics of TO detectors, the TP rate and the FP rate concerning potential misclassifications and FI-campaign–runtime savings, using the miQS benchmark.
Figure 6.6a presents hypothetical FP rates across the runtime of an FI experiment. The percentage of misclassifications significantly decreases after reaching the time t_{gr} mainly because a significant fraction of FI experiments have already terminated (as indicated by Figure 6.3).
Figure 6.6b shows various hypothetical TP rates. The highest savings during a campaign occur at the time of the optimal threshold (t_{gr}) for each rate. However, they never quite reach the savings achieved by the optimal solution (OPT), which means the later a target outcome (TO) is correctly classified, the less overall campaign-runtime savings can be achieved.
The campaign-runtime savings peak at the time of t_{gr} for each rate, but they indeed never reach the level of savings achieved by OPT. This implies that the later a TO is correctly classified, the lower the overall campaign-runtime savings.

Before t_{gr}, an absolute detector has limited utility as many TO experiments have not yet commenced, which shows that it is not a suitable activation point. However, as time progresses, a TO detector loses potential for savings as the $3t_{\text{gr}}$ limit approaches.

Immediately after t_{gr}, even detectors with a TP rate of 70 percent can save approximately 18 percent of the overall FI campaign runtime. When invoked at or after t_{gr}, absolute detectors have a benchmark-specific upper limit regarding potential savings that never reach the OPT's savings.

This upper bound savings for the MiBench benchmarks, when a detector reaches TP = 1, range between 77.7 percent (miBC) and 80.27 percent (miRDE) of the OPT savings; for instance for miRDE, the upper bound is $\frac{\text{TP=1 savings}}{\text{OPT savings}} \triangleq \frac{17.83\%}{22.21\%} \approx 80.27\%$.

In conclusion, the analysis demonstrates that absolute TO detectors, activated *shortly after* t_{gr} where they have minimal negative impact, provide benefits even if they exhibit a high FP rate. At this point, even if the TO detectors are less effective at detecting TO experiments (low TP rate), their potential for saving FI-campaign runtime remains substantial. This observation holds for all the benchmarks under examination.

6.3 Initial Timeout-Detector Research

Now that the potential of TO detectors as acceleration methods for FI campaigns is described, I will sketch two approaches and outline my findings. The findings revealed that these approaches were *ineffective* in reducing the overall campaign duration; consequently, these methods were discontinued in the subsequent stages of my work. Nevertheless, this initial research has provided the necessary knowledge and guided the development of the effective TO detector ACTOR.

6.3.1 Jump-Address Histograms

The IP register contains the memory address of the upcoming instruction to be fetched and executed. I include all non-monotonic IP changes with the term *jump*, including unconditioned jumps and conditioned branches. The dynamic control flow of the program (see Section 5.1.2 on page 101) is extractable with the help of the *objdump* of the program's ELF file and the program trace from the tracing step. This extraction allows us to trace the order in which individual unconditioned and conditioned jumps occur.

My initial approach to creating a TO detector relies on two key assumptions: (1) infinite loops result in TOs, which are detectable by tracking an increased frequency of jump or branch addresses, and (2) that jump addresses unknown to the golden run lead to invalid instructions or unfamiliar control flows. However, in addition to TO detection, assumption (2) can lead to other failure classes, such as traps, due to invalid memory accesses.

The program tracing process in FAIL* is expanded to create a *golden histogram* in addition to the golden run, which records all jump addresses during the tracing and a unique jump-address histogram for each FI experiment. Using a set of metrics, the base concept compares the histograms from each FI experiment with the corresponding golden histogram at the same time. Based on that, the FI experiment is classified as a TO, or the experiment continues the execution.

During the development of this approach, it became evident that collecting data on the intended system state in the form of jump-address histograms is *excessively* data-intensive and impractical to process. The designed *neural network* trained using this vast data. The resulting models were then employed for TO detection within FAIL*.

The abundance of metrics and recorded jump histograms constitutes a *vast* metric space; finding the proper configuration for the TO detector is not trivial. During my student's master thesis, my

student and I opted to analyze this collected data to ensure the correct parameterization of the metrics and decision-making using the neural network through the *Keras* [Ker] deep learning tool. The resulting trained models were used to guide the TO-decision process.

To address the trigger time for the TO detector, FAIL* was extended to set $3t_{\text{gr}}$ *absolute* framework thresholds. This threshold bounds the period between the FI time point t_{p_i} and the framework threshold $3t_{\text{gr}}$ for potential executions of dynamic TO detectors. Initially, I attempted to run the TO detector multiple times, even periodically. However, it quickly became obvious that this approach was infeasible due to the amount of data processed.

Consequently, I conducted initial analyses similar to those in Figure 6.6a, revealing that a significant portion of the experiments terminated after $1.2t_{\text{gr}}$ on average. I also evaluated other thresholds, such as $0.5t_{\text{gr}}$ and $0.8t_{\text{gr}}$, but these thresholds lead to low detector quality (low TP and high FP rates). As a result, I focused solely on the TO detector threshold of $1.2t_{\text{gr}}$; Figure 6.6a shows a more detailed analysis, which confirmed this threshold.

However, applying the models leads to *significant* overhead. None of the benchmarks exhibited a positive effect on the FI-campaign runtimes. Thus, this initial implementation of this approach seems *impractical* overall.

6.3.2 Adaptive Timeout Thresholds

This second approach is inspired by the work of Edwards and colleagues [ESE14]. The authors *adapt* TO thresholds using information obtained before or during the program run and apply this approach to automatically evaluate student assignments. They assume is that if a former test case fails due to a TO, the next one might reach the TO threshold. Therefore, the authors decrease the threshold every time a TO occurs.

Based on this idea, the following approach to reduce the FI-campaign runtime is to *adjust* the TO threshold *during* the campaign's progression using the resulting failure classes of the previous FI.

During my student's master thesis, we hypothesized that if an FI triggers a TO, a TO will likely occur if a *higher significant* bit is injected. Therefore, when a TO occurs, the threshold is decreased for the remaining higher-significance FIs (up to the *Most Significant Bit (MSB)*). In the plain implementation of FAIL*, each job contains FI instructions for one byte. Therefore, it was obvious to use a *local* adjustable threshold byte-wise. The initial threshold for the *Least Significant Bit (LSB)*'s FI was $3t_{\text{gr}}$. If one of the flipped bits leads to a TO, the threshold is *reduced* for the remaining FI experiments of the byte; otherwise, it is not reduced.[40] The reduction value in two versions of the TO detector is 5 percent and 10 percent, resulting in potential reductions of 35 percent and 70 percent.

However, we expected to observe a logarithmic reduction pattern, with significant reductions in the beginning and relatively consistent reductions for the higher bits, given the high probability of TOs in those cases. The evaluation results validate our expectations. The 10 percent approach seems more promising, and the quality of the TO detector remains very high. The precision[41] and accuracy[42] of the results are consistently above 98 percent. This high level of precision and accuracy

[40] The initial concept reduces the threshold for each occurring TO and subsequently increasing the threshold for a non-TO, with the constraint that the upper limit of $3t_{\text{gr}}$ is not exceeded. For the initial exploration of adaptive thresholds, the threshold has only been reduced for every TO.

[41] During the evaluation of the approach, various binary classification ratios have been used. In addition to the TP and FP rates mentioned earlier, the *precision* describes the ratio of the number of TPs to the total number of FI experiments classified as TO. The precision is calculated as $\frac{\#TP}{\#TP+\#FP}$.

[42] The *accuracy* is determined by considering all correct classifications (TP and TN) in relation to the total number of FI experiments: $\frac{\#TP+\#TN}{n}$.

is reflected in the TP and FP rates, suggesting a high quality of prediction. For the 5 percent version, the quality is even better.

However, the overhead of this approach is negligible, as it involves only counting the number of TOs and adjusting the threshold in the corresponding callback of FAIL*. Considering the end-to-end runtime of the campaigns, the reductions vary from only 1.39 percent (miRDD) to 3.1 percent (miBFE), with miSHA as an outlier at 5.4 percent.

At this point, we assumed it would be more like guessing parameterizations (e.g., reduction value, locality of the threshold) rather than relying on real system states. Consequently, we discontinued this approach because it resulted only in *minor savings*.

6.4 Autocorrelation Timeout Detection

In this section, I introduce the dynamic TO detector ACTOR. To understand how ACTOR works, I will first explain the concept of autocorrelation in the context of TOs based on the work of Ibing [IKP16]. Ibing and colleagues describe an on-the-fly TO detector for software that records jump addresses and makes decisions based on *autocorrelation* to determine whether the running program ends in a TO. I will discuss the aspects we have used as the concept for ACTOR and delve into the parameterization of this *heuristic*, explaining the specific parameters used in ACTOR and the reasons behind their selection.

6.4.1 Autocorrelation

Autocorrelation is a technique commonly used in signal processing and statistical analysis. Ibing and colleagues [IKP16] used autocorrelation to detect periodic infinite loops. The authors apply discrete autocorrelation to the *jump-history list* rather than the entire program trace. They do so because the sequence of jump targets is sufficient to reconstruct the complete path through the (dynamic) program's structure.

The autocorrelation considers last m jumps at a particular time and compares the recorded jump sequence with a time-lagged version of itself. The discrete autocorrelation can be simplified into a recursive count function $a(g,m)$, with g representing the current *lag*:

$$a(g,m) = \begin{cases} a(g,m-1)+1 & \text{if } H(m) = H(m-g) \\ 0 & \text{else} \end{cases} \tag{6.3}$$

In a nutshell, we count how many times the program jumps g backward in time periodically, starting from the last taken jump ($H(m)$), before encountering a jump target that is different from $H(m)$. This process is repeated with different lags (e.g., $g \in [1,64]$) on a fixed jump-history list H, resulting in a vector of autocorrelation count values, denoted as $\vec{a}$.

For instance, in the visualization of Figure 6.7, the program is stuck within an endless loop that takes alternating conditional jumps with each iteration, visualized by a CFG (see Section 5.1.1 on page 101) and the jump-history list, which is also a segment of the program's *dynamic control flow* (see Section 5.1.2 on page 101). Since the last jump target $H(m)$ is taken every four jumps, we end up with an autocorrelation vector of $\vec{a} = (0,0,0,4)$. If the autocorrelation value exceeds a given threshold a_{thr}, it classifies a TO.

We adapted this concept for TO detection within the FI context and successfully implemented it in FAIL*, resulting in the dynamic TO detector called *ACTOR*.[43]

[43] ACTOR stands for *AutoCorrelation-based TimeOut Restriction*.

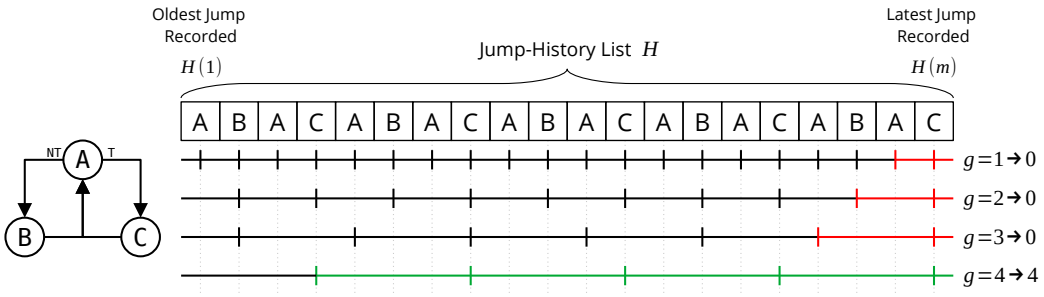


Figure 6.7 – Exemplary Autocorrelation Timeout Detection.
On the left is a minimalistic example of a CFG with BBs `A`, `B`, and `C` (see Section 5.1.1 on page 101). During the program's execution, the program flow jumps to different addresses, recorded in the jump-history list H with the predefined length m in the order they occur. Starting with the most recently recorded jump, denoted as $H(m)$, this approach counts whether and how often the address $H(m)$ reappears within the lag interval for a series of different lags g. If a counter exceeds a predefined threshold a_{thr}, the running program is classified as a TO.

6.4.2 Implementation

In this section, I will discuss the individual parameters of the autocorrelation approach and how they are adapted in the context of FI and the implementation of ACTOR. The core functionality of ACTOR was implemented as a separate FAIL* plugin and in a separate FAIL* campaign definition. This plugin simultaneously creates a golden-run jump-history list containing all the golden run's sequential jumps when the golden run is generated in the initial FAIL* tracing step. The plugin is executed during every FI experiment using this golden-run jump-history list.

To implement autocorrelation as an absolute, dynamic TO detector in FAIL*, we need to make adaptations and select the *four* parameters for ACTOR, including: (1) Firstly, it is essential to determine when and how often the autocorrelation is triggered, which means the dynamic TO *detector threshold*, as well as when the *recording* of the jump-history list H starts. The ACTOR-specific parameters are (2) the *jump-history–list length* m, (3) the set of *lags* $g_1, g_2, \ldots$ used for counting the jump addresses that occur, both necessary for Equation 6.3 and (4) the *jump-counter thresholds* $a_{\text{th},1}, a_{\text{th},2}, \ldots$ to determine if the SUT's execution can be classified as a TO or not, after applying different lags for the recursive counting function a.

In cases where ACTOR classifies the FI experiment as non-TO, the framework threshold $3t_{\text{gr}}$ serves as a fallback to catch potential FNs. The main challenge with this approach is managing the TO detector's overhead and detection latency, as these factors are critical for achieving actual end-to-end savings while influencing the quality of ACTOR.

6.4.2.1 Timing of ACTOR

Ibing and colleagues [IKP16] focused on natively run programs and used the binary-instrumentation package *Pin* [Luk+05] to hook *all* jumps of the programs under evaluation. They continuously ran the autocorrelation on every jump from the program's start and used a jump-history length 100. This approach resulted in slowdown factors of 100 to 225, which would negate any potential savings achievable with a TO detector.

Considering the lessons learned from my initial attempts, as mentioned in Section 6.3, and the workload analysis outlined in Section 6.2.3, ACTOR diverges from Ibing's method in two significant points: (1) Recording jumps during the execution of an FI experiment has a noticeable impact on the

performance of any FI framework. To mitigate this, ACTOR initiates jump recording *after* t_{gr}, where active FI experiments decrease rapidly (Figure 6.3). (2) ACTOR records jumps until the jump-history list reaches a specific length, m, and then performs the autocorrelation *exactly once*, minimizing the overhead for each FI experiment but may result in missing some TOs. If ACTOR does not detect a TO, no additional overhead occurs.

The preliminary workload analyses from Section 6.2.3, particularly the insights visualized in Figure 6.6a, and the findings from the jump-histogram approach (as discussed in Section 6.3.1), led to the adoption of ACTOR's dynamic threshold $1.2t_{\text{gr}}$.

6.4.2.2 Jump-History–List Length

The jump-history length m is determined based on the analysis of post-FI instructions, as shown in Figure 6.3. Given that the TO detector triggers at $1.2t_{\text{gr}}$, we calculate the size of the jump-history list m by considering the remaining runtime factor after t_{gr} (which is 0.2 with the dynamic threshold $1.2t_{\text{gr}}$), the benchmark's average jump density (d), and the length of the golden run t_{gr}.

For example, if we have a benchmark in which every tenth instruction is a jump, and the golden run t_{gr} contains 1000 instructions, we set the jump-history–list length m to $0.2 \cdot \frac{t_{\text{gr}}}{d} = \frac{1000}{10} = 20$ branches.

Due to potential deviations in the injected SUTs compared to the correct execution, the jump-history list may be filled before or after $1.2t_{\text{gr}}$ but still near $1.2t_{\text{gr}}$.

6.4.2.3 Set of Lags

The following parameter we need to determine is the set of lags for ACTOR. We used all lags ranging from 1 to a predefined maximal lag $g_{\max}$: $g_i \in [1, g_{\max}]$.

The choice of $g_{\max}$ impacts ACTOR's sensitivity to patterns with different periodicities. A larger $g_{\max}$ allows ACTOR to detect patterns with more extended periods, but it may lead to more brittle decisions, resulting in a higher FP rate.

For example, if we set $g_{\max}$ to 128, ACTOR could detect periodic sequences that repeat only every 128 branches. However, for such cases, a valid TO detector decision would require a jump-history list length m that is quite huge, potentially increasing the FI experiment runtime.

To balance sensitivity and practicality, we chose a maximum lag $g_{\max} = 16$, aligning with our aim of detecting tight loops.

6.4.2.4 Jump-Counter Threshold

The recursive count function from Equation 6.3, denoted as a, takes a lag as input. Since different lags and different distances between jump comparisons are involved, different maximum counters are possible. This variation makes setting a single counter threshold for all lags impossible. Consequently, a separate counter threshold $a_{\text{th},i}$ must be defined for each lag g_i.

Ibing et al.[IKP16] did not have restrictions on the jump-history–list length m, allowing them to use a relatively large counter threshold (e.g., 500) that applied uniformly across all lags. However, with ACTOR's fixed-sized jump-history list, $a(g, m)$ is always less than $\lfloor \frac{m}{g} \rfloor$, making lag-specific counter thresholds necessary.

This necessity creates a vector of counter thresholds, denoted as $\vec{a} = (a_{\text{th},1}, a_{\text{th},2}, \dots)$. The FI experiment is classified as a TO if any observed value exceeds its threshold.

To calculate the individual counter thresholds, we require the *entire* jump history of the golden run, denoted as H_{gr}, during the tracing process. The single counter thresholds $a_{\text{th},i}$ are computed in the tracing step as follows:

$$a_{\text{th},i} = 1 + \max_{j \in [0,(|H_{\text{gr}}|-m)]} a(g_i, H_{\text{gr}}[j, j+m])$$

To calculate $a_{\text{th},i}$, we identify the maximum value for the counting function a that we observe when performing autocorrelation on the golden run. ACTOR shifts an m-sized window over H_{gr} and calculates the autocorrelation to achieve this. With the resulting vector $\vec{a}$, ACTOR cannot trigger if confronted with a regular program run, even if the FI shifts the execution beyond t_{gr}.

6.4.3 Evaluation

After introducing ACTOR and its parameterization, I will discuss the evaluation *results* of ACTOR, implemented in FAIL*. First, I will explain the procedure and rehearse the *ground truth* for the evaluation. Then, I will delve into the specific results that reflect the *effectiveness* of this method. I will discuss the overall runtime *savings* and the decision *quality*, as it is an essential aspect of the TO detector's effectiveness, as described in Section 6.2.2. The technical setup of this evaluation is the same as described in Section 3.2.3.2 on page 69

6.4.3.1 Evaluation's Ground Truth

The $3t_{\text{gr}}$ threshold proved to be a practical compromise (compared to the original > 100 runtime factor) between the overall FI campaign runtime and the high probability of detecting TOs correctly compared to OPT (see Section 6.2.3.2). The ground truth for each benchmark consists of the pilots according to the DUP method (see Section 2.3.2.1 on page 47) and the corresponding number of n_{DUP} FI experiments.

To assess the *effectiveness* of this FI-experiment acceleration method, as detailed in Section 3.2.1.3 on page 62, all FI-experiment runtimes (meaning the total number of SUT's executed instructions) are recorded with the framework threshold set at $3t_{\text{gr}}$, along with the corresponding experiment failure classes. Additionally, we measured the end-to-end FI-campaign runtimes for each FI campaign, from the very start of the first pilot's execution to the completion of the last pilot, including all overheads incurred by FAIL*.

6.4.3.2 Results

Through our evaluation, we aim to demonstrate that ACTOR effectively reduces the end-to-end FI-campaign runtimes without biasing the result statistics toward correct TO classifications. We present the theoretical optimum OPT would achieve if invoked at every FI time t_{p_i} and compare ACTOR's decisions to those of a static TO detector invoked at the same time as ACTOR, which is $1.2t_{\text{gr}}$.

For our evaluation, we focused on the MiBench benchmarks of the benchmark portfolio (see Section 3.2.3.1 on page 67) but excluding the Micro benchmarks. I exclude the Micro benchmarks because their runtimes are too short to produce statistically stable and significant results.

We compared ACTOR against OPT and a static $1.2t_{\text{gr}}$ detector in the evaluation. OPT is a benchmark for the upper bound of campaign-runtime savings, whereas the $1.2t_{\text{gr}}$ is a static (overheadless) alternative to ACTOR.

Timeout-Decision Quality

First, let us examine the impact of ACTOR on the ratios of the occurring failure classes (see Table 6.2, Misclassification Δ%). Overall, we observe that ACTOR has a minimal effect on the determined failure classes from the ground truths across all benchmarks, shifting less than 0.5 percent of all FIs from one failure class to a TO. Additionally, ACTOR's threshold at around $1.2t_{\text{gr}}$ successfully limits the *invocation* of ACTOR to less than 10 percent of all experiments (see Table 6.2, Inv.), except for miSHA, which experiences a high number of long-running timeouts, as shown in Figure 6.2a.

Furthermore, the ACTOR detector is highly effective, with a TP rate exceeding 85 percent in aborting FI experiments that would continue its execution until $3t_{\text{gr}}$. In all cases, the static $1.2t_{\text{gr}}$ detector, invoked around the same time as ACTOR, reclassifies more experiments as TOs as ACTOR.

Benchmark	Misclassification [Δ%]				$1.2t_{\text{gr}}$	ACTOR		
	Ben.	SDC	Trap	TO	TO [Δ%]	Inv.	TP Rate	FP Rate
miBC	+0.00	−0.02	+0.00	+0.02	+0.04	4.69 %	98.51 %	93.40 %
miBFD	−0.07	−0.13	+0.00	+0.20	+0.28	8.21 %	96.76 %	84.12 %
miBFE	−0.06	−0.12	−0.01	+0.19	+0.31	8.37 %	98.27 %	82.40 %
miQS	−0.05	−0.35	−0.03	+0.43	+1.41	9.01 %	88.55 %	37.72 %
miRDD	−0.07	−0.08	−0.03	+0.19	+1.49	8.51 %	85.65 %	5.27 %
miRDE	−0.03	−0.26	−0.11	+0.40	+0.46	5.63 %	99.92 %	56.39 %
miSHA	+0.00	−0.16	−0.11	+0.27	+13.75	27.50 %	99.75 %	42.29 %

Table 6.2 – Quality of the Timeout Detection.
This table presents data comparing the resulting ACTOR and the $1.2t_{\text{gr}}$ detector classification. Under *Misclassification* Δ%, the table shows the percentage changes in the failure classes compared to the ground truth, and under $1.2t_{\text{gr}}$, the change in the TO ratio using the static $1.2t_{\text{gr}}$ detector. The table on the right shows some key data from ACTOR, including the number of ACTOR invocations (Inv.) in relation to the total number of active experiments at $1.2t_{\text{gr}}$ and the TP and FP rates of ACTOR.

However, the FP rate of ACTOR varies significantly, ranging up to 93.4 percent (miBC), which means that ACTOR incorrectly identifies FI experiments as TOs when they would eventually result in a different failure class before $3t_{\text{gr}}$. Despite this variation, we still achieve good results in terms of *misclassification* for the FI experiments that have *not yet* terminated at $1.2t_{\text{gr}}$ for two main reasons: (1) We invoke the detector on only a tiny subset of FI experiments (see Table 6.2, Inv.). (2) From these invoked FI experiments, only a small subset will yield a non-TO result (e.g., see Figure 6.3). As a result, even a high FP rate leads to relatively minor changes in the overall results. I will return to the FP rate issue in more detail in Chapter 7.

Campaign-Runtime Savings

In Table 6.3, I present the runtime savings achieved by ACTOR through the early termination of the FI experiments. ACTOR reduces the FI experiment runtime by a minimum of 13.1 percent (miBC and miRDD) and up to 30 percent (miSHA) for the number of simulated post-FI runs. Compared to OPT, ACTOR achieves a significant reduction despite (on average) invoking $0.7t_{\text{gr}}$ instructions later than OPT (i.e., the FI times t_{p_i} of all FI experiments across all benchmarks).

In direct competition with a $1.2t_{\text{gr}}$ detector, which acts around the same time as ACTOR, ACTOR stays within 5 percent points of the savings achieved by the $1.2t_{\text{gr}}$ detector (except for miSHA).

Benchmark	ACTOR			Post-FI Instr. [%]			End-to-End
	m	$\overline{g}$	$t_{\mathrm{ov}_{\mathrm{ACTOR}}}$	ACTOR	$1.2t_{\mathrm{gr}}$	OPT	
miBC	2 705	3.9952	64 µs	−13.1	−17.3	−25.2	−12.66 %
miBFD	981	9.5484	65 µs	−15.8	−20.4	−28.4	−15.72 %
miBFE	1 019	9.5679	67 µs	−16.1	−20.7	−28.7	−15.91 %
miQS	1 385	1.6200	53 µs	−19.5	−23.3	−32.3	−16.07 %
miRDD	794	1.0094	38 µs	−13.1	−14.5	−20.1	−12.24 %
miRDE	851	1.0070	47 µs	−17.2	−16.0	−22.2	−17.59 %
miSHA	1 766	1.0002	68 µs	−30.6	−44.4	−61.7	−27.64 %

Table 6.3 – ACTOR Quantifications and Overall Campaign-Runtime Savings.
This table quantifies the autocorrelation, including the history-list length m, the average lag $\overline{g}$, and the method overhead of ACTOR $t_{\mathrm{ov}_{\mathrm{ACTOR}}}$. The subsequent section of the table illustrates the reduced percentage of post-FI instructions using ACTOR, the $1.2t_{\mathrm{gr}}$ detector, and OPT as an upper-bound reference. The table also demonstrates the end-to-end FI campaign runtime savings, considering all potential overheads from both FAIL* and ACTOR compared against the only use of the framework threshold $3t_{\mathrm{gr}}$.

ACTOR sometimes achieves more significant runtime savings than the $1.2t_{\mathrm{gr}}$ detector, as executions stuck in tight loops can fill the jump-history list before $1.2t_{\mathrm{gr}}$ (the list length is limited, as described in Section 6.4.2.2).

The jump-history list length m, determined during the tracing step, spans a range from 794 (miRDD) to 2705 (miBC) for the MiBench benchmarks. As described in Section 6.4.2.2, higher list lengths are caused by a determined higher jump density during the benchmark's execution. The average lag $\overline{g}$ of an FI campaign results from considering all the lags from ACTOR invocations that lead to a TO classification. Lower $\overline{g}$ values, as observed in miRDD, miRDE, and miSHA with round about $\overline{g} \approx 1$, indicate that a single, consistently repeating jump address was detected more frequently. Higher $\overline{g}$ values, like for miBFD and miBFE with $\overline{g} \approx 9.5$, suggest that longer jumps occur periodically.

These reductions translate into tangible end-to-end savings (see Table 6.3, End-to-End) for the overall FI campaign runtime, ranging from a minimum of 12.2 percent (miRDD) to as much as 27.6 percent (miSHA). These end-to-end savings represent the overall *effectiveness* of ACTOR as an FI-experiment acceleration method, as defined in Section 3.2.1.3 on page 62.

This success is attributable to two key design decisions: (1) The autocorrelation is *highly efficient*, taking less than 70 µs to execute (see Table 6.3, $t_{\mathrm{ov}_{\mathrm{ACTOR}}}$). ACTOR achieves this efficiency constraining the number of considered lags (see Section 6.4.2.3) and the length of a jump-history list (see Section 6.4.2.2) and invoking the autocorrelation *exactly once*. (2) Recording the jump-history list within the FI experiment results in a 28 percent reduction in the simulation frequency of Bochs because the FI framework delays the simulation when recording. Although this reduction is only active in the interval $[1.0, 1.2]t_{\mathrm{gr}}$, the simulation-time savings during the FI experiment translate effectively into end-to-end FI-campaign–runtime savings.

6.4.3.3 Summary of ACTOR

Inspired by the work of Ibing and colleagues [IKP16], who introduced an autocorrelation-based TO detection method, I present ACTOR [▷Tho+22]. ACTOR is an autocorrelation-based dynamic TO detector for the simulation-based FI-framework on the ISA-layer FAIL*, designed to identify highly-

periodic jump patterns and terminate FI experiments early, thus preventing them from reaching the framework threshold $3t_{gr}$.

Applied to seven benchmarks from the MiBench benchmark suite, ACTOR achieves significant end-to-end savings ranging from 7.4 percent to 27.6 percent compared to a static timeout detector triggered at $1.2t_{gr}$. Notably, ACTOR maintains a low classification error of less than 0.5 percentage points and a high TP rate exceeding 85 percent for experiments.

However, ACTOR exhibits a high FP rate. The bad FP rates relate to the relatively low number of ACTOR invocations during the already low number of active FI experiments at the ACTOR threshold around $1.2t_{gr}$. I will further discuss this topic in Chapter 7.

6.5 Summary of Early Timeout-Detection Mechanisms

In this chapter, similar to the previous chapters 4 and 5, the primary focus is reducing the overall runtime of FI campaigns $t_{cpn} = n \cdot t_{exp}$ through the implementation of acceleration methods. However, unlike chapters 4 and 5, the emphasis here is not on reducing the number of pilots n but on reducing the individual *runtime of FI experiments* t_{exp}.

In this chapter, I delve into a detailed analysis of the workload of the MiBench benchmark FI campaigns. This analysis particularly highlights the resulting failure classes and the number of executed instructions in the individual FI experiments. It becomes evident from this investigation that the potential for savings in FI experiments classified as *Timeouts (TOs)* is substantial, potentially reaching up to 32.26 percent of the executed instructions post-FI time. In the case of miSHA, this potential is even higher at 61.67 percent. Although previous failure-class prediction methods focused on predictions related to SDCs (see Section 2.3.3 on page 49), TOs got less attention in the context of FI campaign acceleration. Hence, developing an FI-experiment acceleration method for early TO detection is the central topic of this chapter.

Therefore, it is essential to discuss the typical approach to managing timeouts: *TO detectors*, whether absolute or relative, static or dynamic, provide the logic to make the corresponding TO *decision* when they trigger at a specific *threshold*. However, TO decisions can result in *misclassifications*. For example, an FI experiment might be wrongly classified as a TO even when it would not have led to a TO without the TO detector. Thus, when using a TO detector as an FI-campaign acceleration method, the effectiveness of an FI-experiment acceleration method (see Section 3.2.1.3 on page 62) stands for the amount of runtime saved. However, it is also essential to quantify the *quality* of the acceleration method which is where binary classification comes into play.

Throughout my research, I developed three different TO detectors. The first two detectors are based on *jump-address histograms* or *adaptively changeable* thresholds. Unfortunately, these approaches proved to be either inefficient or less effective. Despite the challenges encountered, this research has yielded valuable insights. It has provided a deeper understanding of the system-state requirements for TO detectors, their limits, and the significance of setting the TO threshold at 1.2 times the golden run duration t_{gr}.

However, the third TO detector, inspired by the work of Ibing and colleagues [IKP16], led us to the TO detector ACTOR. ACTOR uses *autocorrelation* for the TO detection based on the jumps in the running FI experiment. I discussed ACTOR regarding its corresponding parametrization, implemented in FAIL*, and evaluated it due to the potential overall FI-campaign–runtime savings and the quality of the TO decisions.

The evaluation of ACTOR demonstrates that the impact of FI experiment misclassifications is minimal, below 0.5 percent, which is an improvement over the static approach at the same threshold $1.2t_{gr}$. Simultaneously, ACTOR maintains a high rate of true positives, consistently exceeding 85

percent. However, the false positive rate is relatively high. Still, this can be contextualized by considering the low number of ongoing FI experiments at the threshold $1.2t_{\text{gr}}$ and the limited number of invocations, a maximum of 9 percent or 27.5 percent in the case of miSHA.

Nonetheless, the ACTOR's FI-campaign–runtime savings are significant. The overhead caused by ACTOR is no more than 68 µs, resulting in overall end-to-end FI-campaign savings ranging from 12.2 to 27.6 percent.

The code of ACTOR implemented in FAIL* is publicly accessible via Zenodo.

TRY IT OUT: Autocorrelation Timeout Detection Using ACTOR in FAIL*

You can access and execute ACTOR through the provided archive, which includes FAIL* for performing FI-experiment TO detection via autocorrelation:

`https://doi.org/10.5281/zenodo.6534708`

7

Discussion and Related Work

> I like to move forward and notice things along the roadside that indicate where I should go.
>
> — Joshua Michael "Josh" Homme (born 1973)

In this chapter, I provide a comprehensive discussion of the contributions made in this dissertation, along with the underlying assumptions, results, and relevant associated research. The *fault model* used in this dissertation is the first topic discussed. The discussion of the fault model focuses on the ISA layer chosen for injection and the single-fault assumption. Each *contribution* is discussed in detail, followed by an examination of the *dissertation's setup*, encompassing the simulated hardware in FAIL* and the evaluations' benchmark portfolio. The chapter concludes by exploring related work within the context of FI-campaign–runtime acceleration methods.

7.1 Fault Model

A *Fault Model (FM)* serves as the foundational assumption for assessing the reliability of an SUT concerning the occurrence of faults, their propagation, and their potential impact on the SUT's behavior. Given that I conceptualized all my contributions based on the described FM, in this section, I will delve deeper into the FM outlined in Section 2.2.4 on page 41 and discuss its value.

Two essential aspects of the FM have been the subject of recurrent discussion within the research community: (1) The ISA layer's selection for FIs, which entails a trade-off between accuracy regarding the reliability assessment of the system and the hardware-model complexity and accessibility. (2) Flipping a single bit on the ISA layer to emulate a propagated fault from the hardware. This aspect also has drawn considerable attention and debate.

7.1.1 ISA-Layer

It is possible to conduct FIs in system layers closer to the physical layer, like the microarchitectural layer or the physical layer itself. The proximity of FIs to the physical layer aligns the results more closely to reality. The higher the system layers for FIs is, the fewer potential FI points are accessible due to layer abstractions. Consequently, FIs conducted in these higher SUT layers only indirectly affect the (partly uncovered) segments of the underlying layers (refer to Section 2.2.4.1 on page 41). However, low-layer hardware models are often not available [Sch16]. Conducting assessments at the ISA layer proves much less time-consuming [Cho+13] and offers more comprehensible targets for FI [SRS14]. Nevertheless, the ISA layer remains controversial, primarily due to the perceived *lack of precision* in estimating system reliability.

Cho and colleagues [Cho+13] have delved into this debate, exploring discrepancies in the accuracy of FI outcomes across various layers, spanning from software to the flip-flop layer. Their investigations reveal substantial *differences* in the occurrence of SDCs, sometimes reaching up to an order of magnitude; at higher layers, the occurrence of SDCs in FI outcomes potentially *rises* [Cho+13; SB19].

Despite the perceived inaccuracies associated with the ISA-layer FI, Cho and colleagues emphasize their utility: although not precise, these outcomes can effectively guide correct design decisions. Thus, even though the results may not be flawless, they remain valuable if they lead to accurate system design choices [Cho+13].

The CLERECO[44] [Koo+14; Val+15; Kad+16; Cha+19; PG21] research group builds on Cho's insights and explores *cross-layer* system evaluations using FI. Their investigations focus on analyzing individual layers and estimating the interconnections between them and the assessments of individual layers.

Papadimitriou and their colleagues [PG21] from the CLERECO group present an extensive study from physical origins [Cha+19] to compare the FI's impact across multiple layers, from the microarchitecture to higher layers.

To accomplish this, they employ standard metrics like AVF and PVF (refer to Section 2.2.2 on page 35) alongside specialized variants, such as the *Hardware Vulnerability Factor (HFV)* and *Software Vulnerability Factor (SVF)*. These metrics allow distinct measurements across various segments/layers of the SUT. Additionally, the authors introduce *fault-propagation models* that embrace diverse error types that are, not visible at the ISA layer, such as corrupted instructions or operands/immediates.

Their research depicts different fault-propagation–model distributions across two different ISAs with distinct microarchitectural implementations within the ARM series. Their examination highlights

[44] CLERECO stands for *Cross Layer Early Reliability Evaluation for the Computing cOntinuum* and is an EU FP7 project.

dissimilarities in fault propagations across different hardware despite employing the same ISA. In this context, the authors formulated the following statement:

> *"Any software modification may alter the hardware access and utilization patterns (and thus, its vulnerability to soft errors) in such a way that the mix of fault manifestations that reach the software layer also changes dramatically. As a result, any effort to reduce the vulnerability of the software which does not consider the vulnerability of the underlying hardware may skew the analysis and lead to pitfalls in design protection." [PG21]*

The authors claim that using higher layers, such as the ISA layer, for FIs is generally too imprecise. For measuring PVF and SVF, the authors use the LLFI tool [TP13; Lu+15] to inject faults into the immediate representation of the SUT's program, which is analogous to FIs on the ISA layer [Pal+19]. I agree with this statement in general, but it is essential to distinguish between the two perspectives of this statement.

Firstly, deriving conclusions about hardware vulnerabilities by executing FIs on the ISA layer, intending to implement hardening techniques in the *hardware*, is a common approach [PG21]. However, Papadimitriou and colleagues discovered that this method leads to erroneous conclusions, which is because fault propagation within the hardware, such as errors in instructions, their operands/immediates, or the escaping (masking) of faults within the hardware itself, may not be accurately traceable or addressable solely at the ISA layers. Hence, focusing solely on the ISA layer overlooks a significant portion of hardware-related fault propagations, as it restricts the consideration of erroneous data exclusively at the ISA layer. The author mentions correctly that *"it is impossible to accurately analyze the vulnerability of a program for hardware errors without having the underlying hardware information of faults-propagation effects"* [PG21].

Secondly, taking a hardware-agnostic perspective to observe FI on ISA layers to make conclusions about SIHFT techniques for hardening the SUT's software on *fixed* hardware offers advantages, as indicated by the authors themselves [PG21]. However, their study reveals a discrepancy between the evaluations based on PVF and SVF metrics and those considering the entire hardware or whole system stack (using HVF or AVF). The authors consider the failure classes *benign*, SDC, and all other cases simultaneously, uniting them into a "crash" failure class. Notably, they highlight that the proportion of SDCs through FI surpasses crashes at software layers compared to lower layers. The authors attribute this fact to the inadequate consideration of a higher layer of the hardware's fault propagation; using a higher layer for FI considers the fault model for addressing incorrect data only.

Moreover, the authors present the outcomes of a case study of two benchmarks to compare occurrences of SDCs in SIHFT-hardened algorithms by the Δ-encoding [KF14] with unhardened variants. Unfortunately, there is no explanation for why the authors chose these two benchmarks for the comparison. Their findings demonstrate a reduction in the relative proportion of SDCs when considering the software layers only (PVF and SVF), as expected through the implemented SIHFT technique. However, they note an overall increase in the vulnerability of the SUT, encompassing both SDCs and crashes, when using the SIHFT-hardened benchmarks, and the whole SUT is considered (AVF). Thus, the authors conclude that FIs conducted at higher layers, such as the ISA layer, yield *inaccuracies* and often result in *opposite* outcomes.

The AVF, a metric applied to individual components, is typically assessed using the arithmetic mean to describe the overall vulnerability of the SUT [Muk+03b; Muk+03a; PG21]. However, Papadimitriou and colleagues take an alternative approach by including structural factors of the hardware, such as cache sizes, to account for varying susceptibilities among different components. In contrast, the authors focus solely on the quantity of hardware *locations*, using it as the basis for comparing all layers. To evaluate comprehensively the program's behavior under FI of the SUT, the consideration of the *temporal* dimension remains essential and is missing in their consideration.

Papadimitriou and colleagues acknowledge the increased execution time resulting from implementing SIHFT techniques, but none of their used and evaluated metrics account for the temporal dimension. The absence of a two-dimensional FS assessment, combined with the authors' exclusive comparison of *relative* proportions of occurring failure classes, further restricts the validity of the results of their analysis.

Conversely, Schirmeier and colleagues [SB19] have previously explored the ISA layer for FI, presenting results that contrast Papadimitriou's findings. The authors build upon Cho's statements [Cho+13] concerning the ISA layer for FIs. Schirmeier's work highlights the effectiveness of the ISA layer for FIs, contingent upon two assumptions. Firstly, the SUT's hardware must remain *fixed* for this evaluation. FIs on the ISA layer estimate the need to improve segments of the SUT that are visible from the ISA layer, thus not the hardware itself. The primary goal is to enhance the SUT's reliability solely through *algorithmic alternatives* or explicitly implemented *SIHFT techniques*, aligning with Papadimitriou's hardware-agnostic view. Secondly, using an appropriate *metric* becomes crucial, considering the *temporal* aspects of the program executed by the SUT.

The initial investigations, spanning from the digital-logic layer to flip-flops and ending at the ISA layer, confirm the previously noted trend: the closer the injected layer is to the software, the greater the presence of SDCs observed within the distributions of failure classes. This work considers 18 diverse benchmarks subjected to FI campaigns, including Δ-encoded [KF14] benchmarks. Using the fault-coverage factor [BCS69; Lev+09] (i.e., the proportion of non-SDC results in all FIs of a campaign, as referenced in Section 2.2.2 on page 35) as a standard evaluation metric, the 18 benchmarks underwent individual *ranking* based on the fault-coverage factor for each examined layer. Notably, most of the benchmarks are search algorithms aiming to identify the *most reliable* search algorithm, irrespective of other requirements; the other benchmarks are partly from MiBench.

Schirmeier uses the Kendall correlation [Ken38] to illustrate that the rankings among layers exhibit significant *dispersion* in the rankings through the layers. Consequently, concluding the reliability of different layers becomes challenging when relying solely on the fault-coverage factor as a metric.

The fault-coverage factor *lacks* information concerning the program's runtime and the specific locations used within the SUT. In contrast to Papadimitriou's research, Schirmeier introduces the *Extrapolated Absolute Failure Count (EAFC)* metric [SB19]. This metric includes the *size* of the FS as a factor for the fault-coverage factor to weight. Thus, EAFC not only considers the size of structures/locations of the FS, akin to Papadimitriou's study but also integrates the *temporal* dimension of the FS.

Rankings based on EAFC reveal strong correlations across layers, except for this work's strongly restricted register-on-write FM. Consequently, conclusions drawn regarding algorithm selection or implementing SIHFT techniques at the ISA layer *hold* validity for lower layers within the same hardware configuration and FM (as described in Section 2.2.4 on page 41).

Papadimitriou's critique about the ISA layer being too imprecise and potentially leading to opposite conclusions holds significance when the dynamics of the SUT are not considered (i.e., the temporal dimension of the FS). Additionally, their findings indicated an overall increase in the SUT's vulnerability, notably in the proportion of crashes. However, they also demonstrated a reduction in the proportion of SDCs by implementing hardening techniques, a trend consistent with Schirmeier's research.

The higher occurrence of crashes, although significant, is not *critical*. Each crash, partitioned in this dissertation as timeouts and traps, is *inherently detectable*, either by reaching a timeout threshold or by the SUT's trap itself. In contrast, SDCs require active intervention in the design process to be detectable, making this failure class particularly the most dangerous one, especially when the executed and expected system output is unknown. Therefore, the increase in potential crashes

resulting from implemented hardening techniques is not adverse regarding error detectability within the system.

Schirmeier's research demonstrates the effectiveness of using the ISA layer for FI on *fixed* hardware, particularly in enhancing reliability through software adjustments, given the use of an appropriate metric such as the presented EAFC metric, especially for assessing SDCs occurrences. Indeed, as Papadimitriou correctly points out, a comprehensive analysis of the entire system offers greater insight, especially in implementing hardware adjustments to improve the system. However, the authors overlook the *complexity* of conducting such analyses across all layers to determine the system's overall vulnerability. Consequently, the practicality of assessing the system's overall vulnerability remains challenging and unclear in their context. Furthermore, the authors do not delve into the input parameters of the used benchmarks, making the value of their statements unclear. Schirmeier and colleagues have published their benchmark source code, which could aid in estimating the practicality.

In conclusion, employing ISA-layer FI proves highly effective for refining *software adjustments* on *fixed* hardware within the SUT, particularly in evaluating implemented SIHFT techniques for hardening against SDCs. Accordingly, applying my contributions to the ISA layer is also valid.

7.1.2 Single-Fault Assumption

In the FM's description of this dissertation (see Section 2.2.4 on page 41), I detailed a systematic approach involving the injection of single-bit flips into the SUT to emulate transient hardware faults or, rather, *Single-Event Upsets (SEUs)* (refer to Section 2.1.2 on page 13). However, during system execution, a single SEU at the physical layer can manifest as multiple errors across higher layers due to propagation effects through these layers. A straightforward illustration of this phenomenon is the multiple accesses to a flip-flop at distinct times or the multiple readings of a bit from various instructions targeting different addresses. Consequently, a single SEU can cause multiple bits visible on the ISA to flip, resulting in what is known as a *Multiple-Bit Upset (MBU)*. The simultaneous occurrence of multiple SEUs is highly improbable to the extent [MW79; Nor96b; Li+07], so I disregard this in my work. Therefore, incorporating a single SEU as a component of the FM suffices for comprehensive consideration.

According to Cho's findings [Cho+13], multi-bit errors are limited to instances where they are *adjacent* to each other if they occur at all. For instance, word-wise multi-bit errors in registers are quite common [Li+10]. The emergence of complex multi-bit patterns is improbable [Cho+13], and realistic multi-FI pattern models are non-existent [SB19]. A viable approach to approximate occurring multi-bit patterns is flipping entire bytes simultaneously, like in FAIL*, with specific parameters[45] [Sch16].

Each occurrence of a multi-bit error is separable into its single-bit errors. As detailed in the described generic FI process (see Section 2.2.2 on page 35) and as it is in FAIL*, I *systematically* traverse the FS, creating distinct FI experiments to cover each (effective) point within the *whole* FS. Consequently, individual FIs within the campaign encompass every potential permutation of locations where a multi-bit pattern might arise. The primary difference is the possibility of *altered* masking effects or system failures when single errors contrast with the occurrence of multiple errors.

Similar to specific metrics such as the FIT rate or the fault-coverage factor, the critical quantitative aspect is whether the SUT behaves correctly and as expected or however not. Ultimately, as considered in my dissertation (refer to Section 3.1.2 on page 56), various failure classes primarily differentiate failures *technically* but do not conclusively determine whether the SUT executes without failure. The

[45] FAIL* has a *burst mode* that makes flipping whole bytes at once possible.

specific types of failures are rather significant in subsequent (qualitative) analyses or optimizations of the FI process.

From this perspective of evaluating the SUT, four potential scenarios exist when repeatedly considering individual bit flips and a multi-bit flip. For the sake of clarity, I contrast two individual bit flips with a multi-bit flip where the equivalent two bits flip simultaneously:

1. If each bit flip is benign and the multi-bit flip also results in benign behavior, this case holds minimal concern as only the externally observable benign behavior is significant.

2. In cases where individual bit flips and the multi-bit flip result in non-benign behavior, the effect of the multi-bit flip on the *system state* probably differs from the two resulting system states of the individual bit flips. Although this distinction may be necessary for qualitative analysis, it remains of secondary significance in determining the *reliability* of the SUT from the perspective of whether bit flips are benign.

3. In scenarios where two individual bit flips result in non-benign behavior but the corresponding multi-bit flip is benign, exploring this multi-bit scenario also holds minimal significance. This means each bit's behavior has masked the other's faulty effects. As a result, this particular case is irrelevant in the analysis of multi-bit errors and can be ignored.

4. More critical is the scenario when individual bit flips result in benign behavior, but the corresponding multi-bit flip results in non-benign behavior. Despite the systematic traversal of the FS through individual single-bit–flip FI experiments, multi-bit flip would be *misclassified* solely based on the benign classification of the individual results. This scenario leads to a severely wrong conclusion about the impact of the multi-bit on the SUT based on individual bit flip results.

As a reminder, the probability of a SEU occurrence is $1.328 \cdot 10^{-11}$ percent following the Poisson distribution [MW79; Nor96b; Li+07] (see Section 2.2.4 on page 41). An SEU is the fundamental requirement for an error to become visible on the ISA layer. Consequently, the probability of an SEU occurrence is the *upper-bound* probability that *at least one* bit flip becomes simultaneously visible on the ISA layer as an SDC. However, this probability calculation excludes considerations of resulting in other failure classes and fault-propagation models inherent in the hardware, as presented in Papadimitriou's work (e.g., escaping faults, erroneous instructions, incorrect operands/immediates) [PG21].

Incorporating a comprehensive hardware model into this calculation is challenging because of the hardware model complexities and the lack of availability of realistic hardware models [SB19]. Thus, I present the upper-bound probability estimation for a multi-bit flip here. Consider a (bitwise) `AND` instruction from the data flow of an x86 processor, exemplary for the miSHA benchmark. Beginning with the probability of an SEU as the base, the subsequent factors are incorporated for an upper-bound estimation for a two-bit flip.

- Upon examining individual bit values, the output does not change for individual bit flips and alters only when *both* bits flip simultaneously. This situation occurs within $\frac{1}{4}$ of the 2^2 value constellations: two *zero* input bit values.

- In the x86 architecture, data words of up to 32 bits are accessed. Considering this word length alongside this instruction, the probability that the i-th bit will flip in one input and the i-th bit in the other input is $\frac{1}{32}$.

- Within the miSHA benchmark, there exist 40 647 instructions. Consequently, the probability that the two bits flip simultaneously in the same `AND` instruction is $\frac{1}{40\,647}$.

Hence, even based on this simple and straightforward calculation, the factor $\frac{1}{4 \cdot 32 \cdot 40\,647} \cdot 100 = 1.92 \cdot 10^{-5}$ (percent) is multiplied by the SEU probability, substantially reducing the probability by five orders of magnitude.

This estimation remains an upper-bound probability when considering only this particular instruction. With each additional instruction in this flip scenario, the upper-bound probability progressively approaches the actual probability. Factors that are unavailable, such as hardware masking effects, would further *lower* the probability. Moreover, the probability of multi-bit flips occurring across different data words is notably reduced [Li+10].

In conclusion, this simple calculation shows that the probability of a multi-bit flip at the ISA layer, where individual errors lead to benign behavior, and the corresponding multi-bit error results in a non-benign behavior, is *extremely low*. As demonstrated in the previous list, considering additional determined factors further reduces the probability,. Consequently, emulating an SEU as a bit flip *suffices* for the reliability analysis of an SUT on the ISA layer.

7.2 Contributed Fault-Injection–Campaign Acceleration Methods

This section provides a detailed exploration of the acceleration methods contributed in chapters 4 to 6. I discuss each method individually and summarize its strengths, weaknesses, and potential future research topics.

7.2.1 Data-Flow–Sensitive Fault-Space Pruning

The *Data-Flow Pruning (DFP)* method [▷PDL21] introduced in Chapter 4 relies on an extracted data flow and formulates FESs by utilizing *deterministic Instruction-Local Fault-Equivalence Set (IFES)* mappings for use on the ISA layer. In comparison to the DUP method, DFP uniquely uses the *location dimension* of the FS. DFP is precise, covers the whole FS, and stands as an *enchanced* version of the DUP, uniting the DUP's FESs into larger ones via additional logic using the data flow and its instructions from the golden run. The derived *Data-Flow–Aware Fault-Equivalence Sets (DFESs)* can be used by sampling or completely and systematically for full FS coverage.

DFP serves as a complete *replacement* for the DUP. In cases where IFES mappings do not exist, DFP determines the equivalent FESs as the DUP. However, implementing any IFES mappings, even if for simple instruction semantics like `MOV`, potentially reduces the number of FIs. Nonetheless, manually implementing such IFES mappings for specific ISAs and instructions is a *risk* of programming errors. It requires a one-time implementation for each instruction, its operand access patterns, and each specific ISA. For instance, within the x86 ISA, different access patterns to registers (e.g., `rax`, `eax`, `ax`, `ah`, `al` for a single register) must be considered.

Hence, the future work is the development of *automated* IFES mapping generators (as sketched in Listing 4.1 on page 82), either explicitly implemented as indicated or determined through processor models such as defined in Sail [Arm+19; Pro23b], for instance. These automated approaches would extract semantics, streamlining the otherwise manual implementation of IFES mappings for diverse ISAs and their single instructions.

Although achieving a pilot reduction of up to 18 percent marks a notable achievement, there remain considerable possibilities for further enhancement. Firstly, expanding the number of IFES

mappings to encompass additional instructions is imperative. Presently, the implemented set of five instructions covers an average of only 51 percent of the program trace's instructions. Secondly, refining the value's lifetime analysis offers the potential for smaller lifetime values (refer to Section 4.2.3.3 on page 85). The current approach adopts an overly *pessimistic* approach by using the overwrite time as the definitive point at which a value becomes inaccessible.

However, as previously suggested, validating that no program flow (even in the presence of a bit flip) can access a value after an inter-instruction fault propagation needs a more sophisticated approach. Achieving this requires an analysis of the binary to deduce all feasible accesses once the propagated fault becomes active. This task is straightforward for registers within the same basic block. However, this analysis becomes rapidly *complex* when values persist across multiple basic blocks or reside in memory, potentially requiring pointer and alias analyses. Nonetheless, finding improvements to the lifetime determination with the conservative overwrite time as the ground truth remains a topic for future research.

Although I claim the precision and completeness of the DFP method, no *formal proof* exists to confirm this. I assume that the straightforward construction of the DFG, as outlined in Section 4.2.1 on page 77, backs this claim by intuition – under the assumption that the considered IFES mappings and the value lifetime estimation are valid. In practice, using solely the FES construction rule for ϵ-nodes (i.e., no IFES mappings exist) demonstrates that DFP constructs the *equivalent* set of pilots as DUP, strengthening this claim. Moreover, extensive experimental validation supports this claim; across $1.42 \cdot 10^{10}$ points in all FSs assessed in the benchmarks, DFP and DUP consistently produce bit-wise equivalent outcomes. Nevertheless, I acknowledge formal proof of DFP's correctness as a relevant topic for future research.

In the previous Section 7.1.2, I highlighted the single-fault assumption underpinning the fault model used in this dissertation. All IFES mappings are developed assuming the presence of *exactly one* bit flip within *one* operand of an instruction. However, assuming multiple bit flips would occur more often due to propagation in the data flow or initially from the propagation through the hardware layers, these bit flips might get *pruned* due to the single-fault assumption within the IFES mappings. This pruning behavior can impact the *accuracy* of the SUT reliability analyses and make them less precise.

For instance, consider the `AND` instruction example mentioned earlier, where both operands hold a bit value of *zero* in the same bit position. In this scenario, one of the two bit positions might not be considered for the resulting IFES, consequently omitting its consideration in the FI-campaign planning step of the DFP. However, as previously discussed, multi-bit flips are highly improbable. Conducting a comprehensive analysis of DFP under the influence of multi-bit flips is a subject for future research.

Finally, the application of DFP extends *beyond* the ISA layer. For instance, it is possible to implement this method within the digital-logic layer: individual gates representing instructions in the DFP and the voltages on the data lines reflecting the values within the DFG. The IFES mappings are the semantics of the gates (i.e., the truth table). Therefore, with suitable adaptations, DFP is extendable to deeper layers. However, this adaption entails huge DFGs and complex determinations and distinctions of values alongside their respective lifetimes as the condition for symbol propagation. These challenges remain as a subject for future research.

7.2.2 Program-Structure–Guided Approximation of Fault Spaces

In Chapter 5, I presented the concept of *Fault-Space Regions (FSRs)* [▷Pus+19], which aims at *approximately* significantly reducing the number of necessary FIs by effectively covering the whole FS. This method involves creating *borders* deduced from program structures – such as *Basic Blocks*

(BBs) and function scopes – segmenting the temporal dimension of the FS. Pilots within the FESs of the DUP that cross such a border are FIs within the FI campaign. The weights of the remaining FESs are distributed proportionately to the used pilot weights for the region, leading to a weighted allocation.

Section 5.3.3.3 on page 116 presents the outcomes of applying the FSR method on the register-only FS. The implementation resulted in substantial reductions in FIs but also caused significant deviations in result accuracy. This stemmed from an increased *bypass* of the injected bit flips from registers to the memory, causing it to exit the register scope if the resulting FES has not crossed a region border.

The degree of bypassing is notably *dependent* on the ISA. In the case of the x86 processor used in this dissertation, known for its limited count of 8 general-purpose registers, a compiler is forced to swap results into memory due to the high register pressure. Consequently, bit flips may exit the register-only FS scope if a region border is between these swapping operations. Thus, the effectiveness of the FSR method regarding the register-only FS heavily relies on the ISA. I *assume* that systems with a larger register count may encounter fewer issues when considering a register-only FS. Optimizing the precision of FSRs or adjusting region border rules for the register-only FS by leveraging architectural hardware properties stands as future research.

In essence, developing region-border rules is not strictly confined to dynamic sources; other sources, not inherently dynamic, are feasible for this purpose. For instance, *static* jump addresses can serve as alternatives instead of relying solely on dynamic BBs. However, purely static approaches encounter limitations due to the absence of dynamic attributes, like indirect jump addresses, which remain unknown during compile time. Exploring the combination of static program information with dynamic data from the trace and evaluating the optimization achieved by such enhanced border rules is a topic for future exploration, as well as how effective static-only approaches are.

The application of FSRs in this dissertation requires a *trace* at all times. However, when considering the implementation of FSR in continuous systems for FI, a fixed trace could not be available due to the infeasibility of achieving deterministic, reproducible re-execution of the SUT. However, achieving complete coverage or analysis of the FS becomes impossible because of the non-finite and ever-expanding nature of the temporal dimension within the FS of continuous systems.

Another limitation when employing region-border rules is when program structures essential for FSR application are generally unavailable, resulting in insufficiently defined region borders. For instance, in the miSHA benchmark and using the *Call Region (CR)* rule, the lack of functions prevents the establishment of helpful region borders (refer to Table 5.3 on page 115). Similarly, the BB rule's effectiveness decreases if the compiler generates predicated instructions or predominantly linear code with significantly fewer jumps.

Moreover, *compiler optimizations* influence the number of regions overall. For instance, loop unrolling executed aggressively by the compiler expands BBs. However, in such cases, the BB rule only considers the data flow flowing directly into and out of the program, neglecting individual loop iterations. As mentioned, static compiler information can be considered to refine this aspect.

Additionally, another critical factor is the *workload* of the program under evaluation. As exemplified for the Micro benchmarks in the evaluation (refer to Section 5.3.2.2 on page 112), applying FSRs is not feasible, mainly when the program workload is low, and the FS is segmented into broader regions, such as with the CRs rule or compiler optimizations like loop unrolling as mentioned before. Consequently, this does not yield a *statistically significant* number of regions and their corresponding weights, thereby preventing the attainment of reliable outcomes within this approximate approach. Investigating the minimum program workload needed for the application of FSRs or instead exploring the lower bound ratio of regions to program workload stands as a subject for future research.

Irrespective of the compiler setup or ISA, FSRs are applicable *beyond* the ISA layer. For instance, flip-flops serve as locations, and clock cycles define time units. The generation of FESs and determining borders for the FSRs are derivable from the flip-flop values within a respective trace. Expanding to higher layers beyond the ISA layer is also feasible. In cases where a language interpreter executes a program, variables function as locations, and statements represent the transitions between distinct points in time of the FS. In such scenarios, the program structure or any components used for establishing a border rule can be directly accessed by the interpreter, if available.

Finally, I will discuss the validation of the FSR described in Section 5.3.5 on page 119. The overarching objective is to demonstrate that applying FSR leads to FI-campaign outcomes equivalent to the ground truth or instead to those obtained through the application of DUP. In contrast to the DFP, which unites FESs into larger ones, using FSR entails a form of *sampling* DUP's FESs. This sampling redistributes the weights of unused inner FES to the pilots of the outer FES. Consequently, the validation concerning DUP is comparatively less critical than with DFP forming new FESs.

However, a border rule is defined to set a border after each system state. In that case, it generates equivalent FESs of size two and larger than those FESs after applying DUP. FESs of size 1 exist as inner FESs within the respective FSRs and, thus, are not injected. Despite this, a minor adjustment allows FESs of size 1 to function as pilots in an FI campaign, ensuring the creation of equivalent pilots overall. Given that DUP does not further optimize the resulting FESs of size 1, this slight adjustment remains legitimate for validation purposes.

7.2.3 Early Timeout-Detection Mechanisms for Fault-Injection–Experiment Acceleration

In this section, I revisit the concept of timeout detection in FI, aiming to distinguish the ACTOR detector [▷Tho+22] from other timeout-detection methods detailed in this dissertation. I delve deeper into several previously mentioned aspects, expanding the discourse with new points and highlighting areas for potential future work.

The determination regarding whether to abort an FI experiment due to a TO or to allow its continued execution (even beyond $3t_{\mathrm{gr}}$) represents a balancing act between the overall campaign runtime and the quality of the TO detector's results. When dealing with programs lacking a strict, definitive endpoint, no TO detector can effectively discern between non-terminating programs and faulty programs that might execute for an extremely protracted yet limited period. Consequently, in general terms, there is *no absolute* ground truth (refer to Section 6.2.3.1 on page 133) for TO detection; instead, there is only a semblance of similarity among different TO detectors. Hence, the campaign designer faces the critical decision of determining whether the observed level of uncertainty is acceptable within the current design phase.

7.2.3.1 ACTOR: Autocorrelation-based Timeout Restriction

In Section 6.4 on page 137, I described ACTOR, a *Timeout (TO)* detector within the context of individual executed FI experiments. ACTOR uses the trace to identify specific parameters to ACTOR's functionality. After parametrization and during the execution of an FI experiment, the experiment records *jump addresses* and analyzes them for jump patterns using *autocorrelation* [IKP16]. If, based on the definition in ACTOR, a noticeable pattern is identified at time $1.2t_{\mathrm{gr}}$, the FI experiment is classified as an TO at an early stage. Otherwise, if no such pattern is recognizable, the experiment normally proceeds.

Using ACTOR, we have successfully achieved earlier detection of non-timeouts, thereby saving FI-experiment runtime. ACTOR achieves experiment runtimes comparable to those of a static $1.2t_{\mathrm{gr}}$

detector, yet getting timeout rates similar to a static $3t_{\mathrm{gr}}$ detector. This performance distinction is best observable in the miSHA benchmark, where we detected 13.8 percent fewer timeouts than the static $1.2t_{\mathrm{gr}}$ detector, concurrently reducing the overall campaign runtime by 27.6 percent.

A viable option for campaign designers is to employ ACTOR with a predetermined static framework threshold, such as the $3t_{\mathrm{gr}}$ in the example of this dissertation. Notably, ACTOR offers flexibility regarding its triggering mechanism, enabling it to activate beyond $1.2t_{\mathrm{gr}}$ if the TO rates surpass predefined safety standards during its use, for instance.

Another aspect for discussion is the considerable variation in the FP rate of ACTOR (refer to Table 6.2 on page 141), which exceeds partly 80 percent. These FPs stem from ACTOR's detection principle, primarily identifying *strongly* periodic jump addresses. TOs are a consequence of injecting a loop counter that influences the execution of loop passes: LSB flips minimally affect the loop pass count, whereas MSB flips cause intermittent alterations in loop passes. Although both types of FI result in periodic branch patterns detected by ACTOR, some experiments conclude before $3t_{\mathrm{gr}}$.

From our perspective, the classification of these FIs as TOs at $3t_{\mathrm{gr}}$ appears somewhat arbitrary, as these experiments might still terminate as an SDC or benign result after $3t_{\mathrm{gr}}$. For assessment purposes, the ground truth relies on $3t_{\mathrm{gr}}$, aligning with outcomes in related literature. However, the FP rates are inconsequential for the investigated benchmarks, given that only a tiny proportion of even low experiments are misclassified as TOs.

In earlier stages of our studies, we explored hardened variants (with triple-modular redundancy) for the MiBench benchmarks. These variants entail running the program thrice consecutively, comparing their results, and determining the outcome through a voting mechanism. Notably, 86 percent of these experiments conclude *before* $3t_{\mathrm{gr}}$. Although this scenario shows a similar program extension as the injection of the loop counter, ACTOR effectively discerns between the third execution and the SUT's outcome, showing a lower FP rate of only 0.59 percent. Hence, we conclude that ACTOR demonstrates a commendable detection quality for evaluating benchmarks with hardening techniques.

One limitation of using ACTOR is its *dependence* on a program trace, which defines the values for ACTOR's parameters jump-history list H and autocorrelation-count vector $\vec{a}$. Future research could explore alternative methods to determine these parameters, potentially enabling the use of ACTOR without relying on a trace. Additionally, although the value for the maximum lag parameter $g_{\max} = 16$ proved effective for our benchmarks, SUTs executing nested loops and conditional jumps may benefit from a higher value for $g_{\max}$. Exploring these adjustments could enhance ACTOR's performance in such scenarios.

A potential *threat* to the validity of ACTOR is our selection of benchmarks primarily, from the automotive and security branches of MiBench. Whereas these benchmarks are generally the best representatives of safety-critical systems in that context, they focus on performance and repetitive behaviors, lacking in complexity regarding jump targets. Additionally, our evaluation of ACTOR is limited to the application at the ISA layer. However, ACTOR's applicability extends to other layers, given that a mechanism exists to record jump targets (e.g., branch-target buffer). Further research across a broader spectrum of benchmarks and system layers could offer deeper insights into ACTOR's effectiveness and adaptability.

7.2.3.2 Initial Timeout-Detector Research

In Section 6.3 on page 135, I outlined my initial research and developed two approaches for TO detection during an FI experiment. Although both methods yielded rather underwhelming outcomes, exploring these approaches provided *valuable insights* crucial for the knowledge about TO detection and the development of ACTOR. One method involves comparing *histograms* derived from recorded

jump addresses (refer to Section 6.3.1 on page 135), and the other method *adapts* static thresholds of an experiment based on the resulting system failures from other experiments (refer to Section 6.3.2 on page 136).

I will sketch additional aspects regarding these two approaches and highlight potential future research.

Jump-Address Histograms

Using the previously recorded trace of the SUT, I conduct counting all occurring *jump addresses* to construct a *golden jump histogram*. The developed FAIL* plugin records separate histograms at different time intervals during individual FI experiments, comparing them to the golden histogram using various histogram-comparing metrics.

The vast volume of data generated from numerous histograms at different intervals makes this approach impractical. However, extensive analysis indicated that the $1.2t_{\mathrm{gr}}$ time point proved effective for triggering TO detection, a conclusion that also *matches* with the findings from developing ACTOR.

Determining the most appropriate metric for histogram comparison is challenging. Established and known and self-developed metrics failed to provide precise, definitive TO classifications. Attempts by my student and myself to leverage machine learning for identifying suitable metrics are not trivial. Consequently, employing an entire spectrum of metrics within a single model is the result. At this point, the research ventured into *data retrieval*, a research area outside the focus of my dissertation. I implemented what I deemed the model's minimal requirement to evaluate the approach's overall capability and limitations.

However, assessing the resulting model during FI-experiment execution incurred substantial overhead, making this approach inefficient and ineffective, as developed so far. Nonetheless, this work significantly contributed to my initial attempts at TO detection.

The histograms serve as a quantitative representation of the entire program's structure. Based on the current findings, there is potential for further optimization. Similar to the approach used in ACTOR, the comparison should be triggered exclusively at $1.2t_{\mathrm{gr}}$ and should not rely on any machine learning model. Future research should focus on identifying a precise *metric* or set of metrics required for an effective and efficient comparison. The exploration into machine learning arose due to the extensive data volume, finally identifying the $1.2t_{\mathrm{gr}}$ trigger point. The question remains whether the histogram, which represents the program structure, is adequate for reliable TO detection and future research on a second try.

Adaptive Timeout Thresholds

In FAIL*, pilots are technically grouped based on their respective bytes to inject. The adaptive timeout threshold method applies a *continuous reduction* in the static threshold for all remaining FI experiments of the byte whenever a TO is detected.

My student and I used a linear reduction of 5 or 10 percent during this research, resulting in maximum reductions of 35 or 70 percent, respectively. However, these reductions are not aggressive enough, mainly when only a few bits of the byte caused a TO. Developing a *non-linear* reduction function could address this variability in results and to achieve a more aggressive reduction strategy, which reduces the threshold strongly and flattens the reduction to a higher amount of resulting TOs. Exploring the potential form of such a reduction function and determining the necessary data inputs for its formulation is a prospective area for future research.

However, this method poses an inherent challenge as it interconnects individual FI experiments, preventing the parallelization of each single experiment. Thus, only the *groups* of experiments (in this case, bytes) are parallelizable. This structural change might be beneficial in reducing infrastructural data load in a client-server setup by redesigning FI experiments that do not rely on interactions with the server except the group itself. However, this reduces the overall campaign's *parallelism* by a factor approximately related to the group size (in this case, 8).

One conceivable approach could be a *global* server-side adaptive threshold that transmits to each client conducting a single FI experiment. Assessing the effectiveness of this coarse-grained approach in terms of overall campaign runtime reduction and its feasibility in managing infrastructural workload remains a subject for future research.

An area for future research is the trade-off between the loss of parallelism, the infrastructure workload, and the potential savings in the end-to-end campaign runtime when employing adaptive TO thresholds. This exploration could demonstrate how much parallelism can be sacrificed for runtime efficiency when applying adaptive TO thresholds.

7.3 Experimental Setup

In this section, I delve into the experimental setup, which affects all results presented in this dissertation, examining its advantages and disadvantages. Initially, I focus on the simulated hardware or used hardware model. Subsequently, I delve into the benchmark portfolio, highlighting the specific benchmarks' requirements and challenges for evaluating FI-campaign runtime improvements.

7.3.1 Simulated Hardware

FAIL* [Sch+15], serving as the FI framework for my experiments, offers integrations with diverse processor models (refer to Section 3.1.1 on page 53), providing a choice for the FI hardware target. Bochs was FAIL*'s most well-integrated simulator when I began my work. For consistency in my research, I opted to continue with Bochs and its simulated x86 processor.

The x86 processor, however, might not ideally suit the embedded domain due to its potentially larger *chip size* and instruction set's *complexity*. Typically, smaller processors like one of the ARM Cortex [Arm22] series are preferred due to their lower unit cost and sufficient functionality for intended purposes. Hence, evaluating based on a hardware model representing an ARM Cortex, for instance, would likely mirror market norms more realistically.

Throughout my contributions, I encounter challenges inherent to the x86 architecture. Implementing IFES mappings for the DFP (refer to Section 4.2.2.2 on page 79) using an x86 ISA is potentially more intricate than other ISAs. Integrating IFES mappings for a single instruction demands the *correct* consideration of all access patterns of the operands or immediates.

The syntax inherent in the x86 ISA[46] requires considering a considerably larger number of edge cases per instruction (e.g., operand-access syntax) than ARM instructions. Consequently, implementing IFES mappings for x86 is inherently more *prone* to implementation errors and more *time-consuming* than other processor ISAs with simpler assembly syntax.

Significant result deviations are evident in the FSR evaluation of the register FM (refer to Section 5.3.3.3 on page 116). This discrepancy arises from the high *register pressure* stemming from numerous indirect memory accesses within the instruction set and a smaller register file, characteristics that may differ significantly in other architectures.

[46]This point is independent of the Intel or AT&T syntaxes.

Throughout several sections of this thesis, the Bochs simulator served as the primary processor simulation, employing a basic timing model of one instruction per cycle and assuming wait-free memory access [Sch16]. The selection of this simulator is intentional due to its simplicity. However, FAIL* now offers interfaces capable of integrating more realistic hardware models; for instance, models defined by the ISA-description language Sail [Pro23b] are assimilable into FAIL* [Die+22].

Nonetheless, certain discoveries would not have been achievable with a thorough, accurate, yet slow and complex simulator in FAIL* [Sch16]. A sluggish FI infrastructure would have hindered feasible FS traversing for most workloads, even with DUP or other acceleration methods.

However, the straightforward hardware model within Bochs, lacking a cache hierarchy, somewhat limits the significance of the obtained FI results. Specifically, the outcomes derived from injecting memory bit flips tend to present a pessimistic overapproximation [Sch16]. In a modern cache hierarchy, certain memory bit flips might be masked, for instance, when the CPU writes a cache line back to the main memory.

7.3.2 Benchmarks for Fault-Injection–Campaigns Improvements

Generally, benchmarks serve as a guideline to evaluate the quality of implemented software, providing a statically comparable basis for drawing qualitative conclusions about the software's performance using metrics or in-depth analysis. In the domain of FI for evaluating system reliability, I suppose four use cases exist:

1. Evaluation of an error detection/correction technique
2. Comparison of several error detection/correction techniques for a target SUT
3. Evaluation of an FI technique or fault model
4. Evaluation of FI–campaign runtime improvements

For all these use cases, suitable benchmarks are necessary. Currently, due to the absence of an accepted *FI benchmark suite*, the research community frequently uses *performance* benchmark suites like Dhrystone [Wei84], PARSEC [Bie+08; BL09], and notably, miBench [Gut+01], as used in several works in the FI domain. Primarily designed to assess hardware performance, benchmark suites like MiBench are the common representatives of the embedded benchmarks domain.

These benchmarks possess properties that may not align with the specific needs of the FI use cases. Some of these performance benchmarks are notably simple, often repeating a set of instructions thousands of times. Whereas these benchmarks may offer advantages, particularly in use case 4 (e.g., when using FSRs), they might not pose significant challenges for FI-campaign–runtime acceleration methods. These benchmarks measure various aspects of a system, emphasizing different resources like CPU, cache, or memory usage. However, these tailored aspects are primarily irrelevant when determining the reliability of an SUT using FI, which is a mismatch in the focused purpose.

In use cases 1 and 2, where the focus is on evaluating error-tolerance techniques, the target program is frequently fixed integrated into the system. This integration allows the evaluation of general failure behaviors of an SUT or the comparison of diverse techniques for selection, aiming to identify the most suitable error-tolerance technique for the specific target system running the target program code. Conversely, in use cases 3 and 4, where the target application is either irrelevant to the generalized development or due to fundamental research purposes, the potential variations for designing a benchmark for these use cases are *boundless*.

When conducting performance measurements using benchmarks on actual, often high-speed hardware, it is common in the FI domain to significantly reduce input sizes to ensure manageable

FI-campaign runtimes (i.e., campaign-runtime explosion, refer to Section 2.3.1 on page 45). However, this practice introduces challenges compared with other research that might use the same benchmarks, especially as the input size is often not specified in papers, or entirely new and unique inputs are generated. Additionally, shrinking input sizes is a challenging task; for instance, resizing graph-based inputs or cropping images without care can lead to invalid or semantically irrelevant inputs. Thus, the reduction of input sizes is specific to each benchmark, demanding extensive manual work and specific implementation.

Additionally to the benchmark, several adjustments are necessary for the *environment* and monitoring capabilities of the FI framework. For instance, when using (simulation-based) FI frameworks to mimic an embedded environment, there is a requirement for various infrastructural elements and additional code integration into the system, such as the `stdlib` or all system input and output interfaces. Particularly, executing the benchmark in a bare-metal environment, as in my case with FAIL* using Bochs, demands manually adjusted implementation and manual integration of third-party code, if needed. Moreover, setting signals or symbols manually (as highlighted in Section 3.2.2 on page 64 or Listing 3.2 on page 65) is essential to ensure the interpretability of the SUT within the FI framework.

Nonetheless, these benchmarks generally evaluate the fundamental effectiveness of the implemented FI-campaign runtime improvements. Although real-world benchmarks are desirable for such evaluations, they are optional for such an evaluation, as the benchmarks should make effective FS-traversal assessable. Additionally, as implied above, benchmarks nearer to real-world applications are challenging to implement in FI frameworks and might be infeasible to evaluate due to the high workload of the campaign. However, the structural characteristics of the benchmarks play a role in acceleration methods, such as the recorded jump targets for FSRs and ACTOR or the data flow and its instructions for DFP. Hence, benchmarks mirroring the behavior of real-world systems would naturally offer superior evaluation conditions.

To focus on these complexities, a possible future goal within the FI community should be the development of a *dedicated* FI benchmark suite. This suite would mirror a predefined target domain and possess program structures more closely aligned with real-world applications.

Developing distinct benchmarks emphasizing explicit susceptibility to SDCs or TOs, among other system behaviors, would be particularly valuable, like benchmarks that are inherently resilient to errors (e.g., numerical algorithms) would be particularly valuable. Moreover, variants of these benchmarks in the suite – either end-to-end semantically equivalent (akin to Schirmeier's work comparing different sorting algorithms [SB19]) or hardened benchmarks – would be advantageous. To enhance reproducibility, an *input generator* capable of consistently producing the same – for the benchmarks semantically valid – input under predefined parameters automatically, neglecting the need for manual and time-consuming adjustments.

Furthermore, the benchmarks should offer compatibility for execution under complete operating systems and bare-metal setups, eliminating the necessity for manual benchmark adaptation within the FI framework. The resulting ELF files – compiled with a preset of compiler options – from this suite should ideally encompass pre-configured symbols and inputs, easing seamless integration or simple adaptation across the FI community's frameworks. This standardized approach would improve the reproducibility and ensure *uniformity* in the FI community's results.

Either way, the selection of benchmarks (refer to Section 3.2.3.1 on page 67) *threatens* the validity of the evaluation results, given that evaluations only consider the characteristics of the chosen benchmarks. This inherent challenge remains unsolvable, and it is crucial to be aware of this fact when exploring the outcomes of the executed FI campaign. Considering a different benchmark portfolio might change the statement about the performance of my presented acceleration methods, potentially resulting in other FI-campaign–runtime reductions. Additionally, although the Micro

benchmarks initially proved highly beneficial for evaluating and understanding the DFP, their limited workload makes them ineffective for assessing the effectiveness of the developed acceleration methods.

7.4 Related Work

The challenge of the FI-campaign–runtime *explosion* (refer to Section 2.3.1 on page 45) in the FI domain demands innovative approaches for optimization. Throughout this dissertation, I have mentioned several acceleration methods briefly. This section provides a comprehensive overview of the diverse approaches developed to optimize FI-campaign runtimes. To enhance the context, I will consider my acceleration methods – namely the *Data-Flow Pruning (DFP)* [▷PDL21], the *Fault-Space Regions (FSRs)* [▷Pus+19], and our developed TO detector ACTOR [▷Tho+22] – and relate them to the pertinent related work.

To reduce the FI-campaign runtime $t_{\text{cpn}} = n \cdot t_{\text{exp}}$, there are two primary strategies: reducing the required *number of FI experiments* n but guarantee complete FS coverage, or decreasing the *experiment runtime* per FI t_{exp}. The implementation of my acceleration methods, DFP and FSR, directly addresses this by reducing the amount of the necessary FI experiments n, respectively. ACTOR contributes minimizing the experiment runtime t_{exp} through early TO detection.

To efficiently traverse through a huge FS of an SUT, acceleration methods that minimize the required FI experiments are essential. These methods can be categorized based on their *coverage*, whether they offer complete or partial FS coverage, their *precision*, whether their methodology is precise or approximative, or if they are generic regarding the observed system behavior, whether they focus on a specific failure class only or not.

DUP [Smi+95; GS95; Ben+98b; Ber+02; BP03; Bar+05; Gri+12] (refer to Section 2.3.2.1 on page 47) stands out as a straightforward, basic, generic and precise acceleration method that achieves full FS coverage. Although effective, DUP can still result in long campaign runtimes, raising the need for further pilot reduction. DUP is the foundation for the principles behind DFP and FSRs, using and manipulating the FESs derived from DUP. Moreover, DUP's established nature makes it a solid ground truth for evaluating acceleration methods, especially for DFP and FSRs.

In a manner akin to my data-flow analysis but targeting the entire program rather than a singular execution path, Bartsch and colleagues [Bar+17] use *program netlists* [Sch+13] to pinpoint errors guaranteed to be benign across *all* potential program paths. Their methodology complements DFP; my approach addresses all failure classes while concentrating on an individual execution path, enhancing scalability for extended program runtimes.

Compared to DFP, *SmartInjector* [LT13] unifies multiple FESs into larger (potentially also two-dimensional) FESs but relies on heuristic methodology, resulting in *imprecise* pruning. Their focus is also on the failure class SDCs only, whereas other more relevant failure classifications arise [DSE13; Lu+15]. Conversely, DFP is precise, FS-complete, and operates without focusing on a failure class.

On the gate layer, we [▷Die+18] have proposed error-masking terms to detect benign faults within the first cycle after FI, which was the kickoff for the research into the DFP. *Relyzer* [Har+13] and its application to approximate computing, *Approxilyzer* [Ven+19; Ven+16], analyze multiple golden-run executions to identify errors that might behave similarly across all runs. Like DFP, they utilize bit masking and propagation effects at individual instructions. However, their methodology differs from DFP in three key aspects: (1) Precision: Relyzer predominantly prunes imprecisely, although it partially relies on DUP for detecting equivalences within basic blocks. In contrast, DFP ensures precise FS pruning and could serve as a building block within Relyzer. (2) Operand Sensitivity: Although their techniques (*Constant-based Equivalence* and *Constant-based Masking*)

utilize instruction semantics to prune between input and output bits, they focus solely on instructions involving constant operands (e.g., `LSHIFT %al, 5`), ignoring dynamic operand values observed in the golden run. As immediate instructions are relatively uncommon, their reported improvements are minimal. In contrast, DFP considers both constant and dynamic operands. Additionally, my IFES (refer to Section 4.2.2 on page 77) concept establishes equivalences between one input and one output bit and among multiple input bits, enabling backward-directed forking of equivalences (see `AND` in Figure 4.4 on page 76). (3) Pruning Preconditions: I explore preconditions (single static reader and inaccessible dead values) for the data flow to enable precise, inter-location, instruction-sensitive pruning (via DFP's injection-symbol propagation) based on the single-fault assumption (refer to Figure 4.10 on page 88 and Listing 4.3 on page 88). Thus, their constant-based equivalence and masking method inherently lack precision, disregarding secondary data flows from input operands to other instructions.

GangES [SH+14] executes multiple FI experiments concurrently and identifies matching execution states among these experiments. When two experiments are in the same state, only one finalizes, and its outcome is applied to the other. Nonetheless, their matching mechanism is time-restricted post-FI, making their methods ineffective for experiments that become stuck late.

Alternative sampling techniques [Lev+09; Ram+08] provide an approximate coverage of the FS or prioritize significant faults [Ebr+15]. The *fault-similarity heuristic* [SBS14] applies machine learning on recorded system states to prevent injecting similar faults, but it is not reproducible. Some methods use structural attributes, such as data-structure dependencies [Ebr+16], address bounds [SJK17], or memory states [Har+12b], to identify *similar* faults; this is the most similar to my FSR approach. However, these sampling methodologies do not achieve FS-completeness or precision. In contrast, creating FSRs is flexible, reproducible, and FS-complete; the accuracy depends on the weight function only, and it works independently of the failure classification.

Besides FI-based reliability evaluations, various vulnerability factors [AFK01; AT05; Ebr+16; Fan+16; SK09] integrate data from program execution, program structure, and processor architecture to estimate reliability. Despite this, these factors lack a quantitative classification of specific failure behaviors, failing to offer a complete or accurate depiction of the actual FS. *TRIDENT* [Li+18], segments the program into blocks, and propagates SDC probabilities between these blocks without executing real FI. However, this tool is similar to (e)PVF [SK09; Fan+16] and fails to achieve FS-completeness or precision in its analysis due to considerations around branch probabilities and pruned data dependencies.

Regarding dynamic TO detection during the execution of FI experiments, ACTOR represents a prediction method that makes heuristic decisions regarding the outcome of the ongoing experiment, determining whether it is a TO or not. ACTOR is the first attempt at *dynamic* TO detection specifically designed to inject transient hardware dynamically.

Throughout Chapter 6, autocorrelation [IKP16] has been discussed as the significant means to detect infinite loops; however, the initial approach shows an overhead runtime factor of 100 to 225, making the unmodified autocorrelation approach unsuitable for FI. Another method, *Looper* [Bur+09], employs a satisfiability modulo theory solver to generate non-termination formulas verified at runtime, yet it also suffers from excessive runtime overhead factors reaching up to 10000. Carbin and colleagues [Car+11] pursued a different approach by detecting recurring program states, recording the *whole* program states, and marking an infinite loop if no system-state change is observable between two loop iterations. However, in our FI benchmarks, we noticed that timeouts frequently alter their program state (e.g., due to the decrement of a faulty loop counter).

8

Conclusion

> The beautiful thing about learning is nobody can take it away from you.
>
> — Riley B. "B.B." King (1925–2015)

In this final chapter, I revisit and summarize the key points of my dissertation and provide a concise conclusion of my work.

8.1 Summary

The exploration in the field of *dependable computing* comprehends the emerging threat to hardware design operation known as the *"reliability wall"* [BJS07] or *"soft-error wall"* [Muk08]. As transistor sizes in hardware continue to *shrink* [Com22], *transient hardware faults*, also known as soft errors, *threaten* system reliability. To mitigate the threat of errors during operation, hardware or software often use hardening techniques to detect and correct them. Whereas *Software-Implemented Hardware Fault Tolerance (SIHFT)* techniques may be less performant than their hardware counterparts, they offer greater flexibility, cost-efficiency, and adaptability to the target system [Rei+05b; Kon08; RGF23].

Fault Injection (FI) is one possible measure to assess the effectiveness of these hardening measures in handling errors. By using FI frameworks that execute entire FI campaigns, it is possible to analyze and quantify the SUT for reliability systematically. However, a well-defined *Fault Model (FM)* is essential for a purposeful reliability analysis with FI. In the context of this work, I considered the ISA layer to be the target injection layer within the specified FM.

The *Fault Space (FS)* [Gol+06] concept establishes a comprehensive understanding of all potential transient faults. An FI framework executes the FI campaign, which observes the different system behaviors when FIs run at *every* point in time and at *every* bit location regarding the FS of the SUT. However, systematically traversing *all* possible points in time and bit locations of the SUT is not feasible.

Accelerating an FI campaign is achievable by decreasing the number of FIs *pre-injection* or *dynamically* optimizing individual FI executions. The field of accelerating FI campaigns is an extensive research area and has produced several well-established methods, like *Def/Use Pruning (DUP)* [Smi+95; GS95; Ben+98b; BP03; Bar+05; Gri+12] or checkpointing [Par+00a; Par+00b; Ber+02; SH+14; Par+14], for instance. Despite the significant reduction in runtime achieved by appropriate acceleration methods and their combinations, the overall runtime can still be huge.

To further mitigate extensive FI-campaign runtimes, I introduced new acceleration methods for the FI framework FAIL* [Sch+15] as part of my research. These methods use extracted program structures from the SUT's program trace, such as data flows and dynamic jump addresses. All of my contributions need this previously extracted program trace. Acceleration methods outlined in the three contributions have limited use without a proper trace. Without a trace, the acceleration methods outlined in the three contributions can only be used to a limited extent.

I delved into these contributions in chapters 4 to 6 of this dissertation. To determine the effectiveness of this work my benchmark portfolio contains selected benchmarks from the MiBench benchmark suite [Gut+01] and some self-developed Micro benchmarks. The contributions more in detail, and my findings are as follows:

Contribution 1 *Data-Flow–Sensitive Fault-Space Pruning*

My developed acceleration method, *Data-Flow Pruning (DFP)*, initially constructs a *data-flow graph* from an extracted program trace. The DFP uses implemented *Instruction-Local Fault-Equivalence Set (IFES)* mappings that replicate *masking* and *propagation* effects within individual instructions. Subsequently, the DFP assigns unique *injection symbols* for each bit, which the DFP propagates through the data-flow graph via the implemented IFES mappings. Finally, the DFP plans an FI for every *distinct* injection symbol.

Without implemented IFES mappings, the DFP determines results equivalent to the DUP, making the DFP an *enhanced* version of the DUP. Unlike the DUP, the DFP generates *two-dimensional* equivalences, including information on the bit locations instead of using only

the temporal dimension, which is a structural *unification* of DUP's FESs across time *and* bit locations.

The greater the number of implemented IFES mappings, the greater the benefit compared to the DUP. However, manually implementing IFES mappings is error-prone and requires careful attention from its implementer. Furthermore, the propagation of injection symbols is based on two *conservatively* designed conditions, leaving room for optimization.

I implemented the DFP as a FAIL* tool and evaluated its calculation efficiency and FI-reduction effectiveness. DFP consistently completes the FI calculation in 4 to 38 minutes, a substantial gain compared to the saved hours of FI-campaign runtime. Despite implementing only five IFES mappings for relatively simple instructions `ADD`, `AND`, `OR`, `XOR`, and `MOV`, the DFP further reduces the FIs from the DUP by up to 18.42 percent.

Contribution 2 *Program-Structure–Guided Approximation of Fault Spaces*

The *Fault-Space Regions (FSRs)* approach *divides* the FS into temporal segments defined by *region borders*. In this dissertation, I have chosen dynamic basic blocks and function scopes as borders, designating the entry points of FSRs. The FESs from the DUP are interpreted as data flows that stay within one region or traverse from one region to the next. The latter represents the data flows that propagate errors through the program. Consequently, these FESs are FI representatives for the entire region, and their FI results are *approximately* extrapolated with a *weight function* to the remaining non-injected FESs to achieve *complete* FS coverage.

The concept of FSRs has been implemented as a FAIL* tool and evaluated against the DUP using my benchmark portfolio. FSRs with basic blocks are shorter than those defined by function scopes, unions of individual basic blocks. Function scope FSRs lead to significant reductions and high approximation errors, making them too coarse for FSRs. Additionally, the use of FSRs is more precise for FIs in memory compared to FIs in registers.

The evaluation considers two different weighting functions, revealing that the one based on a weighted average is more accurate. Consequently, the evaluation focuses on FSRs with dynamic basic blocks and the weighted-averaged—alike weighting function.

FSRs are consistently determined in under 4.1 seconds over all considered benchmarks. Basic-block FSRs and the weighted weighting function achievesan FI reduction of up to 77.51 percent with an approximation error of less than 2.09 percent. The FSR approach demonstrates a strong ability to maintain the *locality of results*, particularly concerning the failure class SDC, with a median deviation of 11.1 percent.

Contribution 3 *Timeout-Detection Methods for Fault-Injection–Experiment Acceleration*

SUTs may encounter issues where they do not terminate after an FI, such as flipping a bit of a loop counter. It is possible to detect as early as possible that the SUT would not terminate after an FI. In that case, the period between the point of detection and the set timeout threshold, where the SUT's behavior is automatically classified as timeout, is *eliminable*. In this context, *dynamic timeout detectors* can reduce unproductive time for every FI.

As the final contribution to this dissertation, I delved into the nature of timeouts of individual FI. I presented the results of an analysis of timeouts from an FI campaign perspective, outlined an *upper bound* of potential campaign-runtime savings, and proposed a viable time point for *triggering* timeout detectors.

Furthermore, I shared the outcomes of initial approaches to optimize the FI execution time, leading to the development of the timeout detector ACTOR.

ACTOR uses the program trace and extracts dynamic jump addresses to identify *patterns* in jumps through *autocorrelation*, aiming to detect potential endless loops and to predict timeouts. ACTOR has been implemented as a FAIL* tool with associated plugins. The evaluation of ACTOR has a minimal impact of below 0.5 percent regarding timeout *misclassifications*. Additionally, ACTOR shows a high true-positive rate of approximately 85 percent. However, ACTOR shows a high false-positive rate but is not considered critical in relative terms. Overall, ACTOR needs a maximum of 68 µs for the prediction calculation and achieves end-to-end FI-campaign–runtime savings ranging from 12.2 to 27.6 percent.

Beyond the contributions, I discussed pertinent topics such as the fault model used in this dissertation, the ISA layer as a targeted layer for FI, and the single-fault assumption. Although the ISA layer may lack precision in quantifying the overall reliability of hardware, it proves effective for evaluating SIHFT techniques implemented on *fixed* hardware. The single-fault assumption means that only *one* fault occurs at a time, with only a single error subsequently becoming visible on the ISA layer. I thoroughly discussed this aspect and the evaluation setup and outlined why considering single errors visible on the ISA layer suffices for my research objectives.

Furthermore, I discussed my dissertation's setup, detailing the benchmarks used and the simulated hardware integrated into my technical setup. Choosing appropriate benchmarks to quantify the effectiveness of FI-acceleration methods poses challenges, and I demonstrated the potential issues in general and regarding the dissertation's benchmark portfolio. Although the hardware simulation, an x86 in the Bochs simulator as the FAIL* backend, exhibits realism limitations, it proved adequate for quantifying the effectiveness of my contributions.

8.2 Conclusion

Systematic fault injection *emulates* a system under *transient* environmental effects like radiation and heat. Implemented hardening techniques make systems more reliable and can be implemented in software or hardware. Executing fault injection campaigns *quantifies* a system's reliability within *software-implemented* hardening techniques on *fixed* hardware. However, the execution time of fault injection campaigns can become infeasible when an extensive and detailed statement about the system's reliability is desired. To address this challenge, various acceleration methods aim to either *reduce* the number of injections or speed up individual injections of a campaign. This dissertation delves into optimizing campaign runtimes using extracted *program structures* for application-tailored campaign-runtime acceleration. Through the three distinct contributions, I optimized campaign runtimes and semantically enhanced a well-established acceleration method. In summary, I presented three novel acceleration methods, expanding the repertoire of potential and generally established acceleration methods in this context. These thoroughly evaluated and published contributions offer flexibility and can be combined in various ways, including with established methods beyond the dissertation's scope.

Acronyms

AVF *Architectural Vulnerability Factor* **See:** Section 2.2.2
Related to fault coverage, the AVF measures the vulnerability to a fault in a specific hardware component.

BB *Basic Block* **See:** Section 5.1.1
A BB is a sequence of consecutive and non-branching instructions in a program with a single entry point and exit point.

BBR *Basic-Block Region* **See:** Section 5.2.1.1
A BBR is an instantiation of an FSR based on BBs.

CFG *Control-Flow Graph* **See:** Section 5.1.1
A CFG represents a program's static or dynamic control flow, illustrating the sequence of executed instructions.

CPU *Central Processing Unit*
The CPU is the system's core component, executing instructions and managing the overall operation.

CR *Call Region* **See:** Section 5.2.1.2
A CR is an instantiation of an FSR based on function scopes.

DFES *Data-Flow–Aware Fault-Equivalence Set* **See:** Section 4.1
The DFES of the DFP are unified FESs based on the data flow and instruction semantics of the examined program.

DFG *Data-Flow Graph* **See:** Section 4.2.1
A DFG illustrates the data flow of a program, showing interconnections between data values and instructions.

DFP *Data-Flow Pruning* **See:** Section 4.2
Based on a DFG and implemented IFESs, the DFP constructs DFESs that reduce the number of FIs in the pre-injection analysis step.

DRAM *Dynamic Random-Access Memory*
DRAM is volatile memory that stores data using capacitors, requiring periodic refresh cycles.

DUP *Def/Use Pruning* **See:** Section 2.3.2.1
The DUP calculates FESs and its representative pilots based on write and read times of register or memory accesses.

ELF *Executable and Linkable Format* **See:** Section 3.2.2
ELF is a standardized format for binaries, object code, and shared libraries.

FES *Fault-Equivalence Set* **See:** Section 2.3.2
An FES is a set of faults in a system that leads to equivalent observable system outcome during FI.

FI *Fault Injection* **See:** Section 2.2
FI is a testing technique that injects faults into a system to assess its reliability and to identify potential vulnerabilities.

FIT *Failure In Time* **See:** Section 2.1.4
FIT is a metric to express the expected failure rate of a hardware device, which is the number of failures that can happen per one billion (10^9) hours of operation.

FM *Fault Model* **See:** Section 2.2.4
An FM is a simplified representation to escribing which potential faults, errors, or failures may occur wherein a system and how they behave.

FN *False Negative* **See:** Section 6.2.2.2
In binary classification, a FN is when a predictor incorrectly identifies a positive instance as negative.

FP *False Positive* **See:** Section 6.2.2.2
In binary classification, a FP is when a predictor incorrectly identifies a negative instance as positive.

FS *Fault Space* **See:** Section 2.2.1
A complete FS is defined by a fault model and contains all possible fault injections in time and bit-location dimensions.

FSR *Fault-Space Region* **See:** Section 5.2
An FSR is a temporal segment of the FS defined by region borders for selecting FESs to aim FI reduction in the pre-injection analysis step.

IFES *Instruction-Local Fault-Equivalence Set* **See:** Section 4.2.2
IFESs describe sets of input and output bits of a single instruction which are equivalent regarding the instruction's propagation and masking effects.

IP *Instruction Pointer* **See:** Section 2.2.2, Section 3.2.2
The IP is a register that holds the memory address of the next instruction the CPU will fetch.

ISA *Instruction-Set Architecture* **See:** Section 2.1.3, Section 2.2.4
An ISA defines the instruction set a processor can execute, including the instructions and data manipulation supported by the hardware.

LSB *Least Significant Bit* **See:** Section 6.3.2
This bit represents the lowest-order bit in a numerical value, contributing the least to the overall value.

MBU *Multiple-Bit Upset* **See:** Section 2.1.2, Section 7.1.2
An MBU is a multiple-bit error visible on the ISA layer caused by an SEU.

miBC *MiBench Bitcount* **See:** Section 3.2.3.1
This benchmark computes the bitcount of an array of integers with five distinct methods.

miBFD *MiBench Blowfish Decryption* **See:** Section 3.2.3.1
Blowfish is a symmetric block cipher with a variable key length, and this is the decryption.

miBFE *MiBench Blowfish Encryption* **See:** Section 3.2.3.1
This is analogous to miBFD, but it encrypts instead.

miQS *MiBench Quicksort* **See:** Section 3.2.3.1
This quicksort variant sorts an array of strings.

miRDD *MiBench Rijndael Decryption* **See:** Section 3.2.3.1
Rijndael's block cipher with a variable key length is better known under the abbreviation AES, and this is the decryption.

miRDE *MiBench Rijndael Encryption* **See:** Section 3.2.3.1
This is analogous to miRDE, but it encrypts instead.

miSHA *MiBench SHA1 Checksum* **See:** Section 3.2.3.1
The SHA1 hashing algorithm generates a 160-bit message digest for a given input.

MSB *Most Significant Bit* **See:** Section 6.3.2
This bit represents the highest-order bit in a numerical value, contributing the most to the overall value.

PVF *Program Vulnerability Factor* **See:** Section 2.2.2
Similar to AVF, the PVF measures the vulnerability of an architectural-visible resource on the ISA layer.

SDC *Silent Data Corruption* **See:** Section 3.1.2
SDC is a failure class, representing a terminated system returning undetected erroneous output.

SEU *Single-Event Upset* **See:** Section 2.1.2, excursus on page 17
An SEU is a transient fault in electronics by external factors leading to a temporary change in the state of a transistor, also known as a soft error.

SIHFT *Software-Implemented Hardware Fault Tolerance* **See:** Section 2.1.5, excursus on page 25
SIHFT techniques enhance the system's reliability by implementing software-based error detection or correction techniques.

SRAM *Static Random-Access Memory*
SRAM is high-speed volatile memory using flip-flop circuits as long as power is supplied.

SUT *System Under Test* **See:** Section 2.2.2
The SUT is a system variant. Depending on the FI technique, the environment, hardware, or software must be altered to be usable by an FI framework.

SWIFI *Software-Implemented Fault Injection* **See:** Section 2.2.3.3
SWIFI techniques inject faults into a system's software layer to evaluate reliability.

TN *True Negative* **See:** Section 6.2.2.2
In binary classification, a TN is when a predictor correctly identifies a negative instance.

TO *Timeout* **See:** Section 3.1.2, Section 6.2
A system behavior is classified as a TO when the system does not terminate after a predetermined period.

TP *True Positive* **See:** Section 6.2.2.2
In binary classification, a TP is when a predictor correctly identifies a positive instance.

uFIB *Micro Recursive Fibonacci* **See:** Section 3.2.3.1
This benchmark returns the n-th Fibonacci number.

uLSUM *Micro Iterative Looped Sum* **See:** Section 3.2.3.1
The algorithm performs an iterative summation of a sequence of integers.

uMIX *Micro Mixed Bit-Wise Operations* **See:** Section 3.2.3.1
This benchmark extends μLSUM by bit-wise logical operations.

uQSI *Micro Quicksort Iterative* **See:** Section 3.2.3.1
Inspired by an original MiBench quicksort variant, this benchmark sorts three-dimensional vectors by distance to the coordinate origin iteratively.

uQSR *Micro Quicksort Recursive* **See:** Section 3.2.3.1
This is analogous to μQSI, but it performs the sorting recursively instead.

Mathematical Symbols

a *Autocorrelation Count Function* **See:** Section 6.4.1
The function $a(g, m)$, used by ACTOR, calculates the autocorrelation value for the given lag g and jump-history list length m.

$\vec{a}$ *Autocorrelation Count Vector* **See:** Section 6.4.1, Section 6.4.2.4
The vector $\vec{a} = (a_{\text{th},1}, a_{\text{th},2}, \dots)$ contains different autocorrelation thresholds a_i for different considered lags i.

$a_{\mathbf{thr}}$ *Autocorrelation Threshold* **See:** Section 6.4.1, Section 6.4.2.4
If the autocorrelation value of function $a(g, m)$ exceeds the threshold a_{thr}, the FI experiment is classified as a TO.

b *Single Fault-Space Region border* **See:** Section 5.2
A FSR border $b \in T$ is a point in time of the FS to segment the temporal dimension of the FS.

B *Set of Fault-Space Region borders* **See:** Section 5.2
The set B contains all set borders $b_i \in B$ regarding the used border rule for the given FS.

c *FI-Experiment Failure Classification* **See:** Section 3.1.2, Section 3.2.2
After an FI experiment, the SUT shows a certain behavior, which is categorized into one failure class $c \in C$

C *All possible FI-Experiment Failure Classifications* **See:** Section 3.1.2, Section 3.2.2
C is the set of all in this dissertation considered FI-experiment failure classes.

$\mathscr{E}$ *Effective Fault Space* **See:** Section 2.3.2.1
The FS $\mathscr{F} = \mathscr{E} \cup \mathscr{I}$ is segmented into the effective $\mathscr{E}$ and the ineffective part $\mathscr{I}$ of the FS. Injecting into $\mathscr{E}$ is effective, which means that the SUT results in one of the considered failure classifications C post-FI and is not exclusively benign.

$\mathscr{F}$ *Fault Space* **See:** Section 2.2.1
$\mathscr{F}$ is a two-dimensional space $T \times L$ of all possible points in time and accessible locations of a running system.

g *Lag of ACTOR* **See:** Section 6.4.1, Section 6.4.2.3
The lag g, used in the autocorrelation function $a(g, m)$, stands for the distance at which jumps are checked for equality in the jump-history list H.

H *Jump-History List of ACTOR* **See:** Section 6.4.1, Section 6.4.2.2
ACTOR records jumps during runtime in a list H, which has the length m. Single jumps of the list are denoted as $H(i)$.

$\mathscr{I}$ *Ineffective Fault Space* **See:** Section 2.3.2.1
The FS $\mathscr{F} = \mathscr{E} \cup \mathscr{I}$ is segmented into the effective $\mathscr{E}$ and the ineffective part $\mathscr{I}$ of the FS, whereby injecting faults into $\mathscr{I}$ results eventually in a benign failure classification.

i *Numerical Identifier*
For instance, i marks points in time t_i and locations l_i in the $\mathscr{F}$, or the i-th pilot p_i of an FI experiment of all in an FI campaign, or the i-th job from the FAIL*'s job server, or the i-th FES of the effective FS $\mathscr{E}$.

j *Numerical Identifier*
Analogous to i

k *Numerical Identifier*
Analogous to i

l *Location in the Fault Space* **See:** Section 2.2.1
l describes a single location in the FS $\mathscr{F}$ or the FI location of a single pilot $p_i = (t_i, l_i)$

L *All Locations in the Fault Space* $\mathscr{F}$ **See:** Section 2.2.1
The set L contains all accessible locations $l_i \in L$ in the FS $\mathscr{F}$

m *Length of Jump History List* H **See:** Section 6.4.1, Section 6.4.2.2
The value m defines how many jumps ACTOR records in the jump-history list H for the calculation of the autocorrelation value using the autocorrelation function a.

n *Number of FI Campaign Pilots* **See:** Section 2.3.1, Section 2.3.2.1
n stands for the amount of pilots $n = |P|$ to be executed in an FI campaign.

$n_{\textbf{naive}}$ *Number of naive FI-Campaign Pilots* **See:** Section 2.3.1
Without the application of any acceleration logic, the number of pilots is equal to all possible FIs in the whole FS $n_{\text{naive}} = |\mathscr{F}|$

$n_{\textbf{DUP}}$ *Number of DUP FI-Campaign Pilots* **See:** Section 2.3.2.1
$n_{\text{DUP}} = |P_{\text{DUP}}|$ stands for the amount of calculated pilots for an FI campaign after applying DUP.

$n_{\textbf{DFP}}$ *Number of DFP FI-Campaign Pilots* **See:** Section 4.2.4
$n_{\text{DFP}} = |P_{\text{DFP}}|$ stands for the amount of calculated pilots for an FI campaign after applying DFP.

p *FI Campaign Pilot* **See:** Section 2.3.2.1
A pilot p is a representative of an FES and stands for an FI to be executed at a specific point in time t and location l: $p = (t, l)$

P *All FI Campaign Pilots* **See:** Section 2.3.2.1
The set P contains all pilots $P \ni p_i = (t_i, l_i)$ of an FI campaign.

$P_{\textbf{DUP}}$ *All DUP FI-Campaign Pilots* **See:** Section 2.3.2.1
The set P_{DUP} contains all pilots P of an FI campaign after applying DUP.

$P_{\textbf{DFP}}$ *All DFP FI-Campaign Pilots* **See:** Section 4.2.4
The set P_{DFP} contains all pilots P of an FI campaign after applying DFP.

r *FI-Experiment Result* **See:** Section 3.2.2
FI experiment's single result $r_i = (p_i, c)$ of the FI campaign contains the pilot p_i for unique reference, as well as the resulting SUT behavior categorized in a failure class c.

R *All FI Campaign Results* **See:** Section 3.2.2
The set R contains all single FI experiment results $r_i =\in R$ of an FI campaign.

$R_{\textbf{DUP}}$ *All DUP FI-Campaign Results* **See:** Section 3.2.2
This set R_{DUP} contains all single FI experiment results $r_i =\in R$ of an FI campaign with applying DUP in the pre-injection analysis step.

s *IFES Symbol* **See:** Section 4.2.2.1
Every unique IFES symbol s_i represents an IFES considering the masking and propagation effects of the instruction for a set of bits in the input and output of the instruction.

t *Point in Time in the Fault Space* **See:** Section 2.2.1
This describes a single point in time in the FS $\mathscr{F}$ or the FI time of a single pilot $p_i = (t_i, l_i)$

$t_{\mathbf{cpn}}$ *FI-Campaign Runtime* **See:** Section 3.2
t_{cpn} is the period needed for executing a whole FI campaign consisting of all related FI experiments.

$t_{\mathbf{exp}}$ *FI-Experiment Runtime* **See:** Section 3.2
t_{exp} is the period needed for executing individual FI experiment from the experiment's very start until the experiment reports back the observed system behavior post-FI.

$t_{\mathbf{gr}}$ *Golden-Run Runtime* **See:** Section 6.2.1
t_{gr} stands for the runtime of the reference execution (i.e., the golden run).

t_{p_i} *Pilot's Injection Time* **See:** Section 6.1.1
A pilot $p_i \in P$ injects a fault at the specified FI time t_{p_i}.

$t_{\mathbf{prg}}$ *SUT's Program Runtime* **See:** Section 2.3.1,Section 6.1.1
t_{prg} is a segment of t_{exp} and stands for the pure period needed for the execution of the SUT's program without any method or FI-framework overhead.

$t_{\mathbf{ov}}$ *Runtime Overhead* **See:** Section 2.3.1, Section 3.2.1
t_{ov} is the overhead period in either t_{exp}, when the overhead of individual FI is in focus, or t_{cpn}, when the overhead of the whole FI campaign is meant.

T *All Points in Time in the Fault Space* $\mathscr{F}$ **See:** Section 2.2.1
The set T contains all points in time $t_i \in T$ in the FS $\mathscr{F}$

T_P *All Pilot's FI-Experiment Runtimes* **See:** Section 3.2.1.3
This is the set of the FI-experiment runtimes $t_{\text{exp},i} \in T_P$ for every single executed pilot $p_i \in P$

$T_{P_{\text{DUP}}}$ *All DUP Pilot's FI-Experiment Runtimes* **See:** Section 3.2.1.1
Set of the FI-experiment runtimes $t_{\text{exp},i} \in T_{P_{\text{DUP}}}$ for all DUP pilots P_{DUP}.

w *Weight Function for FES* **See:** Section 2.3.2.1, Section 4.1, Section 5.2
Several weight functions w describe the cardinality of sets containing points of $\mathscr{E}$, like FESs, DFESs, or FSRs.

Bibliography

Own Work

[▷Die+18] Christian Dietrich, Achim Schmider, **Oskar Pusz**, Guillermo Payá-Vayá, and Daniel Lohmann. “Cross-Layer Fault-Space Pruning for Hardware-Assisted Fault Injection.” In: *Proceedings of the 55th Annual Design Automation Conference 2018 (DAC ’18)*. San Francisco, California, USA: ACM Press, 2018. ISBN: 978-1-4503-5700-5/18/06. DOI: 10.1145/3195970.3196019.

[▷PDL21] **Oskar Pusz**, Christian Dietrich, and Daniel Lohmann. “Data-Flow–Sensitive Fault-Space Pruning for the Injection of Transient Hardware Faults.” In: *Proceedings of the 2021 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES ’21)* (Virtual Conference). New York, NY, USA: ACM Press, June 2021, pp. 97–109. DOI: 10.1145/3461648.3463851.

[▷Pus+19] **Oskar Pusz**, Daniel Kiechle, Christian Dietrich, and Daniel Lohmann. “Program-Structure–Guided Approximation of Large Fault Spaces.” In: *2019 24th Pacific Rim International Symposium on Dependable Computing (PRDC’19)*. Washington, DC, USA: IEEE Computer Society Press, 2019. DOI: 10.1109/PRDC47002.2019.00044.

[▷Tho+22] Tim-Marek Thomas, Christian Dietrich, **Oskar Pusz**, and Daniel Lohmann. “ACTOR: Accelerating Fault Injection Campaigns using Timeout Detection based on Autocorrelation.” In: *41st International Conference on Computer Safety, Reliability and Security (SAFECOMP 2022)*. Munich, Germany: Springer-Verlag, 2022. DOI: 10.1007/978-3-031-14835-4_17.

Cited Literature

[ACK70] Frances E. Allen, John Cocke, and Ken Kennedy. “A Catalog of Optimizing Transformations.” In: *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*. ACM. 1970, pp. 1–19.

[AFK01] J. Aidemark, P. Folkesson, and J. Karlsson. “Path-based error coverage prediction.” In: *Proceedings Seventh International On-Line Testing Workshop*. July 2001, pp. 14–20. DOI: 10.1109/OLT.2001.937811.

[AKL12] Algirdas Avizienis, Hermann Kopetz, and Jean-Claude Laprie. *The Evolution of Fault-Tolerant Computing*. Vol. 1. Springer Science & Business Media, 2012. ISBN: 978-3-7091-8871-2.

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. “Fundamental concepts of dependability.” In: *Department of Computing Science Technical Report Series* (2001). URL: https://eprints.ncl.ac.uk/fulltext.aspx?url=55707%2f35D90208-2D34-4C19-BFB5-65E037791AE6.pdf&pub_id=160699&ts=637649754792360989 (visited on 2024-03-28).

[AN97] Z. Alkhalifa and V.S.S. Nair. "Design of a portable control-flow checking technique." In: *Proceedings 1997 High-Assurance Engineering Workshop*. 1997, pp. 120–123. DOI: 10.1109/HASE.1997.648049.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley, 1986. ISBN: 0-201-10088-6.

[AT05] Ghazanfar Asadi and Mehdi Baradaran Tahoori. "An analytical approach for soft error rate estimation in digital circuits." In: *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. IEEE. 2005, pp. 2991–2994. DOI: 10.1109/ISCAS.2005.1465256.

[Aid+01] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. "GOOFI: Generic Object-Oriented Fault Injection Tool." In: *Proceedings of the 31st International Conference on Dependable Systems and Networks (DSN '01)*. Washington, DC, USA: IEEE Computer Society Press, June 2001, pp. 83–88. ISBN: 0-7695-1101-5.

[Alk+99] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection." In: *IEEE Trans. Parallel Distrib. Syst.* 10.6 (June 1999), pp. 627–641. ISSN: 1045-9219. DOI: 10.1109/71.774911.

[All+02] A. Allan, D. Edenfeld, W.H. Joyner, A.B. Kahng, M. Rodgers, and Y. Zorian. "2001 technology roadmap for semiconductors." In: *IEEE Computer* 35.1 (Jan. 2002), pp. 42–53. ISSN: 0018-9162. DOI: 10.1109/2.976918.

[All70] Frances E. Allen. "Control Flow Analysis." In: *ACM SIGPLAN Notices* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479.

[Arl+02] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. "Dependability of COTS Microkernel-Based Systems." In: *IEEE Transactions on Computers* 51 (2002), pp. 138–163.

[Arma] *Arm Cortex-M3 Technical Reference Manual r2p0 - Cotext-M3 instructions*. Arm Ltd. 2010. URL: https://developer.arm.com/documentation/ddi0337/h/programmers-model/instruction-set-summary/cortex-m3-instructions (visited on 2024-03-28).

[Armb] *Arm Cortex-M33 Processor Datasheet*. Arm Ltd. 2020. URL: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Processor%20Datasheets/Arm-Cortex-M33-Processor-Datasheet.pdf?revision=8d50f1dd-f018-4d4f-a5f4-9e1c6cd4c9ea&la=en&hash=B041C79D1C773D22CC9572C2D8FF9E554472F55B (visited on 2024-03-28).

[Arm+19] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. "ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS." In: *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*. Proc. ACM Program. Lang. 3, POPL, Article 71. Jan. 2019. DOI: 10.1145/3290384.

[Arm22] Arm Limited. *Arm® Architecture Reference Manual for A-Profile Architecture*. DDI 0487H.a. Cambridge, England, 2022.

[Ass21] JEDEC Solid State Technology Association. "JESD85A: Methods for Calculating Failure Rates in Units of FITs." In: (2021). URL: https://www.jedec.org/system/files/docs/JESD85A.pdf (visited on 2024-03-28).

[Avi+04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic concepts and taxonomy of dependable and secure computing." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Aug. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[Avi71] A. Avižienis. "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design." In: *IEEE Transactions on Computers* C-20.11 (1971), pp. 1322–1331. DOI: 10.1109/T-C.1971.223134.

[Avi+71] A. Avižienis, G.C. Gilley, Francis P. Mathur, D.A. Rennels, J.A. Rohr, and D.K. Rubin. "The STAR (Self-Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design." In: *IEEE Transactions on Computers* 20.11 (1971), pp. 1312–1321. ISSN: 0018-9340. DOI: 10.1109/T-C.1971.223133.

[BCS69] W. G. Bouricius, W. C. Carter, and P. R. Schneider. "Reliability Modeling Techniques for Self-Repairing Computer Systems." In: *Proceedings of the 1969 24th National Conference*. ACM '69. New York, NY, USA: Association for Computing Machinery, 1969, pp. 295–309. ISBN: 9781450374934. DOI: 10.1145/800195.805940. URL: https://doi.org/10.1145/800195.805940.

[BJS07] Shekhar Borkar, Norman P. Jouppi, and Per Stenstrom. "Microprocessors in the Era of Terascale Integration." In: *2007 Design, Automation & Test in Europe Conference & Exhibition (DATE '07)*. 2007. DOI: 10.1109/DATE.2007.364597.

[BL09] Christian Bienia and Kai Li. "Parsec 2.0: A new benchmark suite for chip-multiprocessors." In: *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*. Vol. 2011. 2009, p. 37. URL: http://www-mount.ece.umn.edu/~jjyi/MoBS/2009/program/02E-Bienia.pdf (visited on 2024-03-28).

[BP03] Alfredo Benso and Paolo Ernesto Prinetto. *Fault injection techniques and tools for embedded systems reliability evaluation*. Frontiers in electronic testing. Boston, Dordrecht, London: Kluwer Academic Publishers, 2003. ISBN: 1-4020-7589-8.

[Bal+03] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. "Fault-tolerant platforms for automotive safety-critical applications." In: *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '03. San Jose, California, USA: Association for Computing Machinery, 2003, 170–177. ISBN: 1581136765. DOI: 10.1145/951710.951734. URL: https://doi.org/10.1145/951710.951734.

[Bar+05] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. "Assembly-Level Pre-injection Analysis for Improving Fault Injection Efficiency." In: *Dependable Computing - EDCC 5*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 246–262. ISBN: 978-3-540-32019-7. DOI: 10.1007/11408901_19.

[Bar+17] Christian Bartsch, Carlos Villarraga, Dominik Stoffel, and Wolfgang Kunz. "A HW/SW Cross-Layer Approach for Determining Application-Redundant Hardware Faults in Embedded Systems." In: *Journal of Electronic Testing* 33.1 (Feb. 2017), pp. 77–92. ISSN: 0923-8174, 1573-0727. DOI: 10.1007/s10836-017-5643-3.

[Bar+90] James H. Barton, Edward W. Czeck, Zary Z. Segall, and Daniel P. Siewiorek. "Fault Injection Experiments Using FIAT." In: *IEEE Transactions on Computers* 39.4 (Apr. 1990), pp. 575–582. DOI: 10.1109/12.54853.

[Bau05] Robert C Baumann. "Radiation-induced soft errors in advanced semiconductor technologies." In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316. DOI: 10.1109/TDMR.2005.853449.

[Ben+00] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. "A C/C++ source-to-source compiler for dependable applications." In: *Proceeding International Conference on Dependable Systems and Networks (DSN '00)*. 2000, pp. 71–78. DOI: 10.1109/ICDSN.2000.857517.

[Ben+01a] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Tagliaferri. "Control-flow checking via regular expressions." In: *Proceedings 10th Asian Test Symposium (ATS '01)*. IEEE Computer Society Press, 2001, pp. 299–303. DOI: 10.1109/ATS.2001.990300.

[Ben+01b] A. Benso, S. Di Carlo, G. Di Natale, L. Tagliaferri, and P. Prinetto. "Validation of a software dependability tool via fault injection experiments." In: *Proceedings Seventh International On-Line Testing Workshop*. 2001, pp. 3–8. DOI: 10.1109/OLT.2001.937809.

[Ben+98a] A. Benso, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. "EXFI: A Low-Cost Fault Injection System for Embedded Microprocessor-Based Boards." In: *ACM Trans. Des. Autom. Electron. Syst.* 3.4 (1998), 626–634. ISSN: 1084-4309. DOI: 10.1145/296333.296351. URL: https://doi.org/10.1145/296333.296351.

[Ben+98b] A. Benso, M. Rebaudengo, L. Impagliazzo, and P. Marmo. "Fault-list collapsing for fault-injection experiments." In: *Annual Reliability and Maintainability Symposium. 1998 Proceedings. International Symposium on Product Quality and Integrity*. 1998, pp. 383–388. DOI: 10.1109/RAMS.1998.653808.

[Ber+02] L. Berrojo, I. Gonzalez, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. Lopez. "New techniques for speeding-up fault-injection campaigns." In: *Design, Automation & Test in Europe Conference & Exhibition 2002 (DATE '02)*. Washington, DC, USA: IEEE Computer Society Press, 2002, pp. 847–852. DOI: 10.1109/DATE.2002.998398.

[Bie+08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications." In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. Toronto, Ontario, Canada: Association for Computing Machinery, 2008, pp. 72–81. ISBN: 9781605582825. DOI: 10.1145/1454115.1454128. URL: https://doi.org/10.1145/1454115.1454128.

[Bin+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "The Gem5 Simulator." In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 1st. Springer, 2006. ISBN: 978-0-387-31073-2.

[Bon64] A. Bondi. "van der Waals Volumes and Radii." In: *The Journal of Physical Chemistry* 68.3 (1964), pp. 441–451. DOI: 10.1021/j100785a001. eprint: https://doi.org/10.1021/j100785a001. URL: https://doi.org/10.1021/j100785a001.

[Bor05] Shekhar Y. Borkar. "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation." In: *IEEE Micro* 25.6 (2005), pp. 10–16. ISSN: 0272-1732.

[Bur08] Australian Transport Safety Bureau. "In-flight upset-Airbus A330-303, VH-QPA, 154 km west of Learmonth, WA, 7 October 2008." In: *Australian Transport Safety Bureau (ATSB), Canberra (Australia). Aviation Occurrence Investigation AO-2008-070* (2008). URL: https://www.atsb.gov.au/publications/investigation_reports/2008/aair/ao-2008-070 (visited on 2024-03-28).

[Bur+09] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. "Looper: Lightweight Detection of Infinite Loops at Runtime." In: *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 2009, pp. 161–169. DOI: 10.1109/ASE.2009.87.

[Bur20] Gavin Malcolm Donald Burtt. "How An Ionizing Particle From Outer Space Helped A Mario Speedrunner Save Time." In: *thegamer.com* (Sept. 16, 2020). URL: https://www.thegamer.com/how-ionizing-particle-outer-space-helped-super-mario-64-speedrunner-save-time/ (visited on 2024-03-28).

[CH02] Yuhua Cheng and Chenming Hu. "Threshold Voltage Model." In: *Mosfet Modeling & BSIM3 User's Guide*. Boston, MA: Springer US, 2002, pp. 65–103. ISBN: 978-0-306-47050-9. DOI: 10.1007/0-306-47050-0_3. URL: https://doi.org/10.1007/0-306-47050-0_3.

[CMV02] P. Civera, L. Macchiarulo, and M. Violante. "A simplified gate-level fault model for crosstalk effects analysis." In: *17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings.* 2002, pp. 31–39. DOI: 10.1109/DFTVS.2002.1173499.

[CRA06] J. Chang, G.A. Reis, and D.I. August. "Automatic Instruction-Level Software-Only Recovery." In: *Proceedings of the 36th International Conference on Dependable Systems and Networks (DSN '06)*. Washington, DC, USA: IEEE Computer Society Press, 2006, pp. 83–92. DOI: 10.1109/DSN.2006.15.

[CS17] Cyril R.A. John Chelliah and Rajesh Swaminathan. "Current trends in changing the channel in MOSFETs by III–V semiconducting nanostructures." In: *Nanotechnology Reviews* 6.6 (2017), pp. 613–623. DOI: doi:10.1515/ntrev-2017-0155. URL: https://doi.org/10.1515/ntrev-2017-0155.

[CS90] E.W. Czeck and D.P. Siewiorek. "Effects of transient gate-level faults on program behavior." In: *Proceedings of the 20rd International Symposium on Fault-Tolerant Computing (FTCS-20)* (Newcastle Upon Tyne, UK). Washington, DC, USA: IEEE Computer Society Press, June 1990, pp. 236–243. DOI: 10.1109/FTCS.1990.89371.

[Car+11] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. "Detecting and Escaping Infinite Loops with Jolt." In: *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*. Ed. by Mira Mezini. Vol. 6813. Lecture Notes in Computer Science. Springer, 2011, pp. 609–633. DOI: 10.1007/978-3-642-22655-7_28.

[Car+95] Joao Carreira, Henrique Madeira, João Gabriel Silva, and João Gabriel Silva. "Xception: Software Fault Injection and Monitoring in Processor Functional Units." In: *Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA '95)* (Urbana-Champaign, Illinois, USA). Sept. 1995, pp. 135–149.

[Cha+19] Athanasios Chatzidimitriou, Pablo Bodmann, George Papadimitriou, Dimitris Gizopoulos, and Paolo Rech. "Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments." In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, June 2019, pp. 26–38. ISBN: 978-1-72810-057-9. DOI: 10.1109/DSN.2019.00018. URL: https://ieeexplore.ieee.org/document/8809532/.

[Che+18] E. Cheng, S. Mirkhani, L. G. Szafaryn, C. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra. "Tolerating Soft Errors in Processor Cores Using CLEAR (Cross-Layer Exploration for Architecting Resilience)." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.9 (Sept. 2018), pp. 1839–1852. ISSN: 0278-0070. DOI: 10.1109/TCAD.2017.2752705.

[Cho+13] Hyungmin Cho, S. Mirkhani, Chen-Yong Cher, J.A. Abraham, and S. Mitra. "Quantitative evaluation of soft error injection techniques for robust system design." In: *Proceedings of the 50th annual Design Automation Conference*. 2013, pp. 1–10. DOI: 10.1145/2463209.2488859.

[Chu36] Alonzo Church. "An unsolvable problem of elementary number theory." In: *American J. of Math.* 58 (1936), pp. 345–363. URL: https://phil415.pbworks.com/f/Church.pdf (visited on 2024-03-28).

[Cin+09] Marcello Cinque, Domenico Cotroneo, Catello Di Martino, Stefano Russo, and Alessandro Testa. "AVR-INJECT: A tool for injecting faults in Wireless Sensor Nodes." In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5160907.

[Com22] International Roadmap Committee. *International Roadmap for Devices and Systems 2022 Update (Executive Summary)*. 2022. URL: https://irds.ieee.org/ (visited on 2024-03-28).

[Con03] C. Constantinescu. "Trends and challenges in VLSI circuit reliability." In: *Micro, IEEE* 23.4 (July 2003), pp. 14–19. ISSN: 0272-1732. DOI: 10.1109/MM.2003.1225959.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd. MIT Press, 2009. ISBN: 978-0-262-03384-8.

[Cor13] Altera Corporation. *Introduction to Single-Event Upsets, White Paper*. Tech. rep. WP-01206-1.0. 2013. URL: https://www.altera.com/en_US/pdfs/literature/wp/wp-01206-introduction-single-event-upsets.pdf (visited on 2024-03-28).

[Cor22] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*. 2022. URL: https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-software-developers-manual-volume-1-basic-architecture.html (visited on 2024-03-29).

[DC+14] Stefano Di Carlo, Paolo Prinetto, Daniele Rolfo, and Pascal Trotta. "A fault injection methodology and infrastructure for fast single event upsets emulation on Xilinx SRAM-based FPGAs." In: *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2014, pp. 159–164. DOI: 10.1109/DFT.2014.6962073.

[DH12] Björn Döbel and Hermann Härtig. "Who Watches the Watchmen? – Protecting Operating System Reliability Mechanisms." In: *Proceedings of the 8th International Workshop on Hot Topics in System Dependability (HotDep '12)* (Hollywood, CA, USA). Berkeley, CA, USA: USENIX Association, 2012. URL: https://www.usenix.org/conference/hotdep12/workshop-program/presentation/D{\"o}bel (visited on 2024-03-28).

[DH14] Björn Döbel and Hermann Härtig. "Can We Put Concurrency Back into Redundant Multithreading?" In: *Proceedings of the 14th International Conference on Embedded Software*. EMSOFT '14. New Delhi, India: Association for Computing Machinery, 2014, pp. 1–10. ISBN: 9781450330527. DOI: 10.1145/2656045.2656050. URL: https://doi.org/10.1145/2656045.2656050.

[DHE12] Björn Döbel, Hermann Härtig, and Michael Engel. "Operating System Support for Redundant Multithreading." In: *Proceedings of the 12th ACM International Conference on Embedded Software (EMSOFT '12)*. EMSOFT '12. Tampere, Finland: Association for Computing Machinery, 2012, pp. 83–92. ISBN: 9781450314251. DOI: 10.1145/2380356.2380375. URL: https://doi.org/10.1145/2380356.2380375.

[DL+12] Domenico Di Leo, Fatemeh Ayatolahi, Behrooz Sangchoolie, Johan Karlsson, and Roger Johansson. "On the Impact of Hardware Faults – An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions." In: *Computer Safety, Reliability, and Security*. Ed. by Frank Ortmeier and Peter Daniel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 198–209. ISBN: 978-3-642-33678-2.

[DMH14] Björn Döbel, Robert Muschner, and Hermann Härtig. *Resource-Aware Replication on Heterogeneous Multicores: Challenges and Opportunities*. 2014. DOI: https://doi.org/10.48550/arXiv.1405.2913. arXiv: 1405.2913 [cs.DC]. URL: http://arxiv.org/abs/1405.2913 (visited on 2024-03-28).

[DSE13] Björn Döbel, Horst Schirmeier, and Michael Engel. "Investigating the Limitations of PVF for Realistic Program Vulnerability Assessment." In: *Proceedings of the 5th HiPEAC Workshop on Design for Reliability (DFR '13)*. Berlin, Germany, Jan. 2013. URL: https://ess.cs.uos.de/danceos/publications/HiPEAC-DFR-2013-Doebel.pdf (visited on 2024-03-28).

[DW11] Anand Dixit and Alan Wood. "The impact of new technology on soft error rates." In: *2011 International Reliability Physics Symposium*. 2011, 5B.4.1–5B.4.7. DOI: 10.1109/IRPS.2011.5784522.

[DeB+12] Nathan DeBardeleben, Sean Blanchard, Qiang Guan, Ziming Zhang, and Song Fu. "Experimental Framework for Injecting Logic Errors in a Virtual Machine to Profile Applications for Soft Error Resilience." In: *Euro-Par 2011: Parallel Processing Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 282–291. ISBN: 978-3-642-29740-3.

[Del97] Timothy J Dell. "A white paper on the benefits of chipkill-correct ECC for PC server main memory." In: *IBM Microelectronics division* 11 (1997), pp. 1–23. URL: https://api.semanticscholar.org/CorpusID:17379018 (visited on 2024-03-28).

[Die+22] Christian Dietrich, Malte Bargholz, Yannick Loeck, Marcel Budoj, Luca Nedaskowskij, and Daniel Lohmann. "SailFAIL: Model-Derived Simulation-Assisted ISA-Level Fault-Injection Platforms." In: *41st International Conference on Computer Safety, Reliability and Security (SAFECOMP 2022)*. Munich, Germany: Springer-Verlag, 2022. DOI: 10.1007/978-3-031-14835-4_14.

[Doe14] Bjoern Doebel. "Operating System Support for Redundant Multithreading." Dissertation. TU Dresden, 2014. URL: https://tud.qucosa.de/api/qucosa%3A28444/attachment/ATT-1/?L=1 (visited on 2024-03-28).

[ED12] Michael Engel and Björn Döbel. "The Reliable Computing Base: A Paradigm for Software-Based Reliability." In: *Proceedings of the 1st International Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)* (Braunschweig, Germany). Lecture Notes in Computer Science. Gesellschaft für Informatik, Sept. 2012.

[ESE14] Stephen H. Edwards, Zalia Shams, and Craig Estep. "Adaptively Identifying Non-Terminating Code When Testing Student Programs." In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. Atlanta, Georgia, USA: Association for Computing Machinery, 2014, pp. 15–20. ISBN: 9781450326056. DOI: 10.1145/2538862.2538926. URL: https://doi.org/10.1145/2538862.2538926.

[Ebr+15] Mojtaba Ebrahimi, Nour Sayed, Maryam Rashvand, and Mehdi B Tahoori. "Fault injection acceleration by architectural importance sampling." In: *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2015 International Conference on*. IEEE. 2015, pp. 212–219. DOI: 10.1109/CODESISSS.2015.7331384.

[Ebr+16] Mojtaba Ebrahimi, Mohammad Hadi Moshrefpour, Mohammad Saber Golanbari, and Mehdi B Tahoori. "Fault injection acceleration by simultaneous injection of non-interacting faults." In: *Proceedings of the 53rd Annual Design Automation Conference*. ACM. 2016, p. 25. DOI: 10.1145/2897937.2898023.

[Ech90] Klaus Echtle. *Fehlertoleranzverfahren*. Berlin, Germany: Informatik Springer, 1990. ISBN: 978-0-3875-2680-5.

[Ent+12] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil. "Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection." In: *IEEE Transactions on Computers* 61.3 (Mar. 2012), pp. 313–322. ISSN: 0018-9340. DOI: 10.1109/TC.2010.262.

[FGI05] Irene Finocchi, Fabrizio Grandoni, and Giuseppe F. Italiano. "Designing Reliable Algorithms in Unreliable Memories." In: *Algorithms – ESA 2005*. Ed. by Gerth Stølting Brodal and Stefano Leonardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–8. DOI: 10.1007/11561071_1.

[FGI07] Irene Finocchi, Fabrizio Grandoni, and Giuseppe F. Italiano. "Designing reliable algorithms in unreliable memories." In: *Computer Science Review* 1.2 (2007), pp. 77–87. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2007.10.001. URL: https://www.sciencedirect.com/science/article/pii/S1574013707000202.

[FGI09] Irene Finocchi, Fabrizio Grandoni, and Giuseppe F. Italiano. "Optimal resilient sorting and searching in the presence of memory faults." In: *Theoretical Computer Science* 410.44 (2009). Automata, Languages and Programming (ICALP 2006), pp. 4457–4470. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2009.07.026. URL: https://www.sciencedirect.com/science/article/pii/S0304397509005003.

[FI04] Irene Finocchi and Giuseppe F. Italiano. "Sorting and Searching in the Presence of Memory Faults (without Redundancy)." In: *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. STOC '04. Chicago, IL, USA: Association for Computing Machinery, 2004, pp. 101–110. ISBN: 1581138520. DOI: 10.1145/1007352.1007375. URL: https://doi.org/10.1145/1007352.1007375.

[FI08] Irene Finocchi and Giuseppe F. Italiano. "Sorting and searching in in faulty memories." In: *Algorithmica* 52.3 (2008), pp. 309–332. DOI: 10.1007/s00453-007-9088-4.

[FPFI09] Umberto Ferraro-Petrillo, Irene Finocchi, and Giuseppe F Italiano. "The price of resiliency: a case study on sorting with memory faults." In: *Algorithmica* 53.4 (2009), pp. 597–620. DOI: 10.1007/s00453-008-9264-1.

[FSK98] P. Folkesson, S. Svensson, and J. Karlsson. "A comparison of simulation based and scan chain implemented fault injection." In: *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*. 1998, pp. 284–293. DOI: 10.1109/FTCS.1998.689479.

[FSS09] Christof Fetzer, Ute Schiffel, and Martin Süßkraut. "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware." In: *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '09)* (Hamburg, Germany). Ed. by B. Buth, G. Rabe, and T. Seyfarht. Heidelberg, Germany: Springer-Verlag, 2009, pp. 283–296. ISBN: 978-3-642-04467-0. DOI: 10.1007/978-3-642-04468-7_23.

[Fan+16] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. "ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis." In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2016, pp. 168–179. DOI: 10.1109/DSN.2016.24.

[Fid+06] André Fidalgo, Manuel Gericota, Gustavo Alves, and José Ferreira. "Using NEXUS compliant debuggers for real time fault injection on microprocessors." In: *Proceedings of the 19th International Symposium on Integrated Circuits and Systems Design (SBCCO '06)*. ACM Press, 2006, pp. 214–219. DOI: 10.1145/1150343.1150397.

[Fle+17] Tino Flenker, Jan Malburg, Goerschwin Fey, Serhiy Avramenko, Massimo Violante, and Matteo Sonza Reorda. "Towards Making Fault Injection on Abstract Models a More Accurate Tool for Predicting RT-Level Effects." In: *2017 IEEE Computer Society Annual Symposium on VLSI* (2017). DOI: 10.1109/ISVLSI.2017.99. URL: https://core.ac.uk/display/86592049 (visited on 2024-03-28).

[Fuc96] Emmerich Fuchs. "An evaluation of the error detection mechanisms in MARS using software-implemented fault injection." In: *Proceedings of the 2nd European Dependable Computing Conference (EDCC '96)*. Ed. by Andrzej Hlawiczka, João Gabriel Silva, and Luca Simoncini. Springer-Verlag, 1996, pp. 73–90. DOI: 10.1007/3-540-61772-8_31.

[GKT89] Ulf Gunneflo, Johan Karlsson, and Jan Torin. "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation." In: *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS-19)*. IEEE Computer Society Press, June 1989, pp. 340–347. DOI: 10.1109/FTCS.1989.105590.

[GS95] Jens Guthoff and Volkmar Sieh. "Combining software-implemented and simulation-based fault injection into a single fault injection method." In: *Proceedings of the 25rd International Symposium on Fault-Tolerant Computing (FTCS-25)*. IEEE Computer Society Press, June 1995, pp. 196–206. DOI: 10.1109/FTCS.1995.466978.

[GT07] Tarak Gandhi and Mohan Manubhai Trivedi. "Pedestrian Protection Systems: Issues, Survey, and Challenges." In: *IEEE Transactions on Intelligent Transportation Systems* 8.3 (2007), pp. 413–430. DOI: 10.1109/TITS.2007.903444.

[GWA79] C.S. Guenzer, E.A. Wolicki, and R.G. Allas. "Single Event Upset of Dynamic Rams by Neutrons and Protons." In: vol. 26. 6. Dec. 1979, pp. 5048–5052. DOI: 10.1109/TNS.1979.4330270.

[Gaw+09] Piotr Gawkowski, Maciej Ławryńczuk, Piotr M. Marusak, Piotr Tatjewski, and Janusz Sosnowski. "On improving dependability of the numerical GPC algorithm." In: *European Control Conference (ECC '09)*. 2009, pp. 1377–1382. DOI: 10.23919/ECC.2009.7074598.

[Gla04] David Glaude. "Electronic Voting Random Spontaneous Bit Inversion Explained." In: *Association Electronique Libre* (2004). URL: https://web.archive.org/web/20070927185155/http://wiki.ael.be/index.php/ElectronicVotingRandomSpontaneousBitInversionExplained (visited on 2024-03-28).

[Gla+15] Michael Glaß, Hananeh Aliee, Liang Chen, Mojtaba Ebrahimi, Faramarz Khosravi, Veit B Kleeberger, Alexandra Listl, Daniel Müller-Gritschneder, Fabian Oboril, Ulf Schlichtmann, et al. "Application-aware cross-layer reliability analysis and optimization." In: *it-Information Technology* 57.3 (2015), pp. 159–169. DOI: 10.1515/itit-2014-1080.

[Gol+06] Olga Goloubeva, Maurizia Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance*. 1st ed. New York, NY, USA: Springer-Verlag, 2006, p. 228. ISBN: 0-387-26060-9.

[Gri+12] Johannes Grinschgl, Armin Krieg, Christian Steger, Reinhold Weiss, Holger Bock, and Josef Haid. "Efficient fault emulation using automatic pre-injection memory access analysis." In: *SOC Conference (SOCC), 2012 IEEE International*. IEEE. 2012, pp. 277–282. DOI: 10.1109/SOCC.2012.6398361.

[Gut+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "MiBench: A free, commercially representative embedded benchmark suite." In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. Dec. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[Gö31] Kurt Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I." In: *Monatshefte für Mathematik und Physik* 38.1 (Dec. 1931), pp. 173–198. ISSN: 1436-5081. DOI: 10.1007/BF01700692. URL: https://doi.org/10.1007/BF01700692.

[HA84] Kuang-Hua Huang and Jacob A. Abraham. “Algorithm-Based Fault Tolerance for Matrix Operations.” In: *IEEE Transactions on Computers* C-33.6 (1984), pp. 518–528. DOI: 10.1109/TC.1984.1676475.

[HAN12] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. “Low-cost program-level detectors for reducing silent data corruptions.” In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’12)*. 2012, pp. 1–12. DOI: 10.1109/DSN.2012.6263960.

[HDL13] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. “dOSEK: A Dependable RTOS for Automotive Applications.” In: *Proceedings of the 19th International Symposium on Dependable Computing (PRDC ’13)* (Vancouver, British Columbia, Canada). Fast abstract. Washington, DC, USA: IEEE Computer Society Press, Dec. 2013, pp. 120–121. DOI: 10.1109/PRDC.2013.22.

[HHJ90] Robert W Horst, Richard L Harris, and Robert L Jardine. “Multiple instruction issue in the NonStop Cyclone processor.” In: *ACM SIGARCH Computer Architecture News* 18.2SI (1990), pp. 216–226.

[HK12] Olof Hannius and Johan Karlsson. “Impact of Soft Errors in a Jet Engine Controller.” In: *Computer Safety, Reliability, and Security*. Ed. by Frank Ortmeier and Peter Daniel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 223–234. ISBN: 978-3-642-33678-2.

[HSR95] Seungjae Han, K.G. Shin, and H.A. Rosenberg. “DOCTOR: an integrated software fault injection environment for distributed real-time systems.” In: *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*. 1995, pp. 204–213. DOI: 10.1109/IPDS.1995.395831.

[Ham50] Richard W Hamming. “Error detecting and error correcting codes.” In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[Har+12a] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. “Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults.” In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’12)* (London, England, UK). Vol. 47. 4. New York, NY, USA: Association for Computing Machinery, 2012, pp. 123–134. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2150990. URL: https://doi.org/10.1145/2248487.2150990.

[Har+12b] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. “Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults.” In: *ACM SIGPLAN Notices*. Vol. 47. ACM. 2012, pp. 123–134. DOI: 10.1145/2189750.2150990.

[Har+13] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. “Relyzer: Application resiliency analyzer for transient faults.” In: *IEEE Micro* 33.3 (2013), pp. 58–66. DOI: 10.1109/MM.2013.30.

[Hen+13] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. “Reliable on-chip systems in the nano-era: Lessons learnt and future trends.” In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2013, pp. 1–10. DOI: 10.1145/2463209.2488857.

[Hil00] M. Hiller. "Executable assertions for detecting data errors in embedded control systems." In: *Proceeding International Conference on Dependable Systems and Networks (DSN '00)*. 2000, pp. 24–33. DOI: 10.1109/ICDSN.2000.857510.

[Hil99] M. Hiller. "Error recovery using forced validity assisted by executable assertions for error detection: an experimental evaluation." In: *Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium*. Vol. 2. 1999, pp. 105–112. DOI: 10.1109/EURMIC.1999.794768.

[Hoe12] Bernd Hoefflinger. "ITRS: The International Technology Roadmap for Semiconductors." In: *Chips 2020: A Guide to the Future of Nanoelectronics*. Ed. by Bernd Hoefflinger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 161–174. ISBN: 978-3-642-23096-7. DOI: 10.1007/978-3-642-23096-7_7. URL: https://doi.org/10.1007/978-3-642-23096-7_7.

[Hof+14a] Martin Hoffmann, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. "Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs." In: *Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14)* (Reno, Nevada, USA). IEEE Computer Society Press, 2014, pp. 230–237. DOI: 10.1109/ISORC.2014.26.

[Hof+14b] Martin Hoffmann, Peter Ulbrich, Christian Dietrich, Horst Schirmeier, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "A Practitioner's Guide to Software-based Soft-Error Mitigation Using AN-Codes." In: *Proceedings of the 15th IEEE International Symposium on High-Assurance Systems Engineering (HASE '14)* (Miami, Florida, USA). IEEE Computer Society Press, Jan. 2014, pp. 33–40. ISBN: 978-1-4799-3465-2. DOI: 10.1109/HASE.2014.14.

[Hof+15] Martin Hoffmann, Florian Lukas, Christian Dietrich, and Daniel Lohmann. "dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel." In: *Proceedings of the 21st IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '15)*. Washington, DC, USA: IEEE Computer Society Press, 2015, pp. 259 –270. DOI: 10.1109/RTAS.2015.7108449.

[Hof16] Martin Hoffmann. "Konstruktive Zuverlässigkeit - Eine Methodik für zuverlässige Systemsoftware auf unzuverlässiger Hardware." Dissertation. Friedrich-Alexander-Universiät Erlangen-Nürnberg, Apr. 2016. URL: https://opus4.kobv.de/opus4-fau/files/7038/DisserationMartinHoffmann.pdf (visited on 2024-03-28).

[Hyd10] Randall Hyde. *The Art of Assembly Language*. No Starch Press, 2010. ISBN: 978-1593272074.

[IEC98] IEC. *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission, Dec. 1998.

[IKP16] Andreas Ibing, Julian Kirsch, and Lorenz Panny. "Autocorrelation-Based Detection of Infinite Loops at Runtime." In: *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech 2016, Auckland, New Zealand, August 8-12, 2016*. IEEE Computer Society, 2016, pp. 368–375. DOI: 10.1109/DASC-PICom-DataCom-CyberSciTec.2016.78.

[ISO18] ISO 26262-6. *ISO 26262-6:2018: Road vehicles – Functional safety – Part 6: Product development at the software level*. Geneva, Switzerland: International Organization for Standardization, 2018.

[JR21] Andrew Ayer Jeremy Rowley Andrew O'Brian. "Yeti 2022 not furnishing entries for STH 65569149." In: *Google Groups* (June 30, 2021). URL: https://groups.google.com/a/chromium.org/g/ct-policy/c/PCkKU357M2Q/?pli=1 (visited on 2024-03-28).

[JWN10] Bruce Jacob, David Wang, and Spencer Ng. *Memory Systems: Cache, DRAM, disk*. Morgan Kaufmann Publishers Inc., 2010. ISBN: 978-0-12-379751-3.

[KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. "Debugging Operating Systems with Time-Traveling Virtual Machines (Awarded General Track Best Paper Award!)" In: *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. 2005, pp. 1–15. URL: http://www.usenix.org/events/usenix05/tech/general/king.html.

[KF14] Dmitrii Kuvaiskii and Christof Fetzer. "Practical Encoded Processing." In: *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. 2014, pp. 335–336. DOI: 10.1109/SRDS.2014.62.

[KI94] Wei-Lun Kao and R.K. Iyer. "DEFINE: a distributed fault injection and monitoring environment." In: *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*. 1994, pp. 252–259. DOI: 10.1109/FTPDS.1994.494497.

[KIT93] W.-I. Kao, R.K. Iyer, and D. Tang. "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults." In: *IEEE Transactions on Software Engineering* 19.11 (1993), pp. 1105–1118. DOI: 10.1109/32.256857.

[KK07] Israel Koren and C Mani Krishna. *Fault-tolerant systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 978-0-12088-5251.

[KKA95] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. "FERRARI: A Flexible Software-Based Fault and Error Injection System." In: *IEEE Transactions on Computers* 44 (1995), pp. 248–260. ISSN: 0018-9340.

[Kad+16] F. Kaddachi, M. Kooli, G. Di Natale, A. Bosio, M. Ebrahimi, and M. Tahoori. "System-level reliability evaluation through cache-aware software-based fault injection." In: *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. Washington, DC, USA: IEEE Computer Society Press, Apr. 2016, pp. 1–6. DOI: 10.1109/DDECS.2016.7482446.

[Kal+15] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, Nikos Foutris, and Dimitris Gizopoulos. "Differential Fault Injection on Microarchitectural Simulators." In: *2015 IEEE International Symposium on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015*. IEEE Computer Society, 2015, pp. 172–182. DOI: 10.1109/IISWC.2015.28.

[Kal+98] Z. Kalbarczyk, G. Ries, M.S. Lee, Y. Xiao, J. Patel, and R.K. Iyer. "Hierarchical approach to accurate fault modeling for system evaluation." In: *Proceedings. IEEE International Computer Performance and Dependability Symposium. IPDS'98 (Cat. No.98TB100248)*. 1998, pp. 249–258. DOI: 10.1109/IPDS.1998.707727.

[Kan+96] G.A. Kanawati, V.S.S. Nair, N. Krishnamurthy, and J.A. Abraham. "Evaluation of integrated system-level checks for on-line error detection." In: *Proceedings of IEEE International Computer Performance and Dependability Symposium (IPDS '96)*. IEEE Computer Society Press, 1996, pp. 292–301. DOI: 10.1109/IPDS.1996.540230.

[Kar+08] Naghmeh Karimi, Michail Maniatakos, Abhijit Jas, and Yiorgos Makris. "On the Correlation between Controller Faults and Instruction-Level Errors in Modern Microprocessors." In: *2008 IEEE International Test Conference*. 2008, pp. 1–10. DOI: 10.1109/TEST.2008.4700613.

[Kar+91] Johan Karlsson, Ulf Gunneflo, Peter Lidén, and Jan Torin. "Two fault injection techniques for test of fault handling mechanisms." In: *Proceedings of the IEEE International Test Conference (ITC'91)*. Oct. 1991, pp. 140–. DOI: 10.1109/TEST.1991.519504.

[Kar+94] Johan Karlsson, Peter Folkesson, Jean Arlat, Yves Crouzet, Günther Leber, and Johannes Reisinger. "Comparison and Integration of Three Diverse Physical Fault Injection Techniques." In: *In PDCS 2: Open Conference*. 1994, pp. 615–642. DOI: 10.1007/978-3-642-79789-7_18.

[Ken38] M. G. Kendall. "A New Measure of Rank Correlation." In: *Biometrika* 30.1/2 (1938), pp. 81–93. ISSN: 00063444. URL: http://www.jstor.org/stable/2332226 (visited on 2024-03-28).

[Ker] *Keras: Deep Learning for Humans*. 2023. URL: https://keras.io/ (visited on 2024-03-28).

[Kon08] Kai Konrad. "Implications of microcontroller software and tooling on safety-critical automotive systems." In: *Presentation at Automotive Electronics and Electrical Systems Forum 2008*. Stuttgart, 2008. URL: https://www.ukintpress-conferences.com/conf/08eac_conf/pdf/day_3/kaikonrad.pdf (visited on 2024-03-28).

[Koo+14] M. Kooli, P. Benoit, G. Di Natale, L. Torres, and V. Sieh. "Fault injection tools based on Virtual Machines." In: *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC '14)*. Washington, DC, USA: IEEE Computer Society Press, May 2014, pp. 1–6. DOI: 10.1109/ReCoSoC.2014.6861351.

[Koo14] Phil Koopman. "A case study of Toyota unintended acceleration and software safety." In: (2014). Presentation Slides. URL: http://www2.ing.unipi.it/~a008669/didattica/FMSS/koopman14_toyota_ua_slides.pdf (visited on 2024-03-28).

[Koo+97] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. "Comparing operating systems using robustness benchmarks." In: *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems*. 1997, pp. 72–79. DOI: 10.1109/RELDIS.1997.632800.

[Kor12] Ingo Korb. "A cross-layer analysis of error impact on the instruction execution in an 8-bit microprocessor." Diploma Thesis. Universität Dortmund, 2012.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, CA, USA). Washington, DC, USA: IEEE Computer Society Press, Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[LB17] Jens Lienig and Hans Bruemmer. *Fundamentals of Electronic Systems Design*. 1st ed. Springer Cham, 2017. ISBN: 978-3-319-55840-0. DOI: 10.1007/978-3-319-55840-0.

[LH07] Aiguo Li and Bingrong Hong. “Software implemented transient fault detection in space computer.” In: *Aerospace Science and Technology* 11.2 (2007), pp. 245–252. ISSN: 1270-9638. DOI: 10.1016/j.ast.2006.06.006. URL: https://www.sciencedirect.com/science/article/pii/S1270963806000800.

[LJ11] Matthew Leeke and Arshad Jhumka. “An automated wrapper-based approach to the design of dependable software.” In: *4th International Conference on Dependability (DEPEND'11)*. IARIA, 2011. URL: https://wrap.warwick.ac.uk/45677/ (visited on 2024-03-28).

[LP+20] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. “The gem5 simulator: Version 20.0+.” In: *arXiv preprint arXiv:2007.03152* (2020). DOI: https://doi.org/10.48550/arXiv.2007.03152.

[LR22] Erich L. Lehmann and Joseph P. Romano. *Testing Statistical Hypotheses*. 4th. Springer, 2022. ISBN: 978-3-030-70578-7. DOI: 10.1007/978-3-030-70578-7.

[LT13] Jianli Li and Qingping Tan. “SmartInjector: Exploiting Intelligent Fault Injection for SDC Rate Analysis.” In: *Proceedings of the International Conference on Defect and Fault Tolerance in VLSI and Nanotechnology Systems(DFT '13)*. IEEE Computer Society Press, Oct. 2013, pp. 236–242. DOI: 10.1109/DFT.2013.6653612.

[Lal97] Parag K Lala. *Digital circuit testing and testability*. Academic Press Inc, 1997. ISBN: 978-0124343306.

[Lap+90] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. “Definition and analysis of hardware- and software-fault-tolerant architectures.” In: *Computer* 23.7 (July 1990), pp. 39–51. ISSN: 0018-9162. DOI: 10.1109/2.56851.

[Lap92] J. C. Laprie. “Dependability: Basic Concepts and Terminology.” In: *Dependability: Basic Concepts and Terminology: In English, French, German, Italian and Japanese*. Ed. by J. C. Laprie. Vienna: Springer Vienna, 1992, pp. 3–245. ISBN: 978-3-7091-9170-5. DOI: 10.1007/978-3-7091-9170-5_1. URL: https://doi.org/10.1007/978-3-7091-9170-5_1.

[Law96] Kevin P. Lawton. “Bochs: A Portable PC Emulator for Unix/X.” In: *Linux Journal* 1996.29es (1996), p. 7. URL: https://dl.acm.org/doi/fullHtml/10.5555/326350.326357 (visited on 2024-03-28).

[Lev+09] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. “Statistical Fault Injection: Quantified Error and Confidence.” In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '09. Nice, France: European Design and Automation Association, 2009, pp. 502–506. ISBN: 978-3-9810801-5-5. DOI: 10.1109/DATE.2009.5090716. URL: http://dl.acm.org/citation.cfm?id=1874620.1874743.

[Li+07] Xin Li, Kai Shen, Michael C Huang, and Lingkun Chu. “A Memory Soft Error Measurement on Production Systems.” In: *Proceedings of the 2007 USENIX Annual Technical Conference* (Santa Clara, CA, USA). Berkeley, CA, USA: USENIX Association, 2007, pp. 1–14. ISBN: 999-8888-77-6.

[Li+10] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. "A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility." In: *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*. 2010. URL: https://www.usenix.org/conference/usenix-atc-10/realistic-evaluation-memory-hardware-errors-and-software-system.

[Li+13] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S. Vetter. "Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM Press, 2013, pp. 1–12. DOI: 10.1145/2503210.2503226.

[Li+18] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai. "Modeling Soft-Error Propagation in Programs." In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2018, pp. 27–38. DOI: 10.1109/DSN.2018.00016.

[Lov+02] M.N. Lovellette, K.S. Wood, D.L. Wood, J.H. Beall, P.P. Shirvani, N. Oh, and E.J. McCluskey. "Strategies for fault-tolerant, space-based computing: Lessons learned from the ARGOS testbed." In: *Proceedings, IEEE Aerospace Conference*. Vol. 5. 2002, pp. 5–5. DOI: 10.1109/AERO.2002.1035377.

[Lu+15] Qining Lu, M. Farahani, Jiesheng Wei, A. Thomas, and K. Pattabiraman. "LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults." In: *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. Aug. 2015, pp. 11–16. DOI: 10.1109/QRS.2015.13.

[Luk+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation." In: *Acm sigplan notices* 40.6 (2005), pp. 190–200. DOI: 10.1145/1064978.1065034.

[MAM84] A. Mahmood, D. Andrews, and E. McCluskey. "Executable assertions and flight software." In: *Digital Avionics Systems Conference (DASC '84)*. 1984, pp. 346–351. DOI: 10.2514/6.1984-2726. URL: https://arc.aiaa.org/doi/abs/10.2514/6.1984-2726 (visited on 2024-03-28).

[MBC10] Paul D. Marinescu, Radu Banabic, and George Candea. "An extensible technique for high-precision testing of recovery code." In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, p. 23.

[MT95] G. Miremadi and J. Torin. "Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection." In: *IEEE Transactions on Reliability* 44.3 (1995), pp. 441–454. DOI: 10.1109/24.406580.

[MV12] Wassim Mansour and Raoul Velazco. "SEU fault-injection in VHDL-based processors: A case study." In: *13th Latin American Test Workshop, LATW 2012, Quito, Ecuador, April 10-13, 2012*. IEEE Computer Society, 2012, pp. 1–5. DOI: 10.1109/LATW.2012.6261258.

[MW79] T.C. May and Murray H. Woods. "Alpha-particle-induced soft errors in dynamic memories." In: *IEEE Transactions on Electron Devices* 26.1 (Jan. 1979), pp. 2–9. ISSN: 0018-9383. DOI: 10.1109/T-ED.1979.19370.

[Mad+94] Henrique Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. "RIFLE: A general purpose pin-level fault injector." In: *Proceedings of the 1st European Dependable Computing Conference (EDCC '94)*. Ed. by Klaus Echtle, Dieter Hammer, and David Powell. Springer-Verlag, 1994, pp. 197–216. ISBN: 978-3-540-58426-1. DOI: 10.1007/3-540-58426-9_132.

[Man+11] Michail Maniatakos, Naghmeh Karimi, Chandra Tirumurti, Abhijit Jas, and Yiorgos Makris. "Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller." In: *IEEE Transactions on Computers* 60.9 (2011), pp. 1260–1273. DOI: 10.1109/TC.2010.60.

[McP07] J.W. McPherson. "Reliability trends with advanced CMOS scaling and The implications for design." In: *Proceedings of the IEEE 2007 Custom Integrated Circuits Conference (CICC '07)* (San Jose, CA, USA). Sept. 2007, pp. 405–412. DOI: 10.1109/CICC.2007.4405763.

[Mez+15] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field." In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2015, pp. 415–426. DOI: 10.1109/DSN.2015.57.

[Mic+05] S.E. Michalak, K.W. Harris, N.W. Hengartner, B.E. Takala, and S.A. Wender. "Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer." In: *IEEE Transactions on Device and Materials Reliability* 5.3 (Sept. 2005), pp. 329–335. ISSN: 1530-4388. DOI: 10.1109/TDMR.2005.855685.

[Mir+92] G. Miremadi, J. Harlsson, U. Gunneflo, and J. Torin. "Two software techniques for on-line error detection." In: *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*. Vol. 22. 1992, pp. 328–335. DOI: 10.1109/FTCS.1992.243568.

[Mit14] Sparsh Mittal. "A survey of techniques for improving energy efficiency in embedded computing systems." In: *International Journal of Computer Aided Engineering and Technology* 6.4 (2014), pp. 440–459. DOI: 10.1504/IJCAET.2014.065419. URL: https://www.inderscienceonline.com/doi/abs/10.1504/IJCAET.2014.065419 (visited on 2024-03-28).

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[Muk+03a] S.S. Mukherjee, C.T. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor." In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 2003, pp. 29–40. DOI: 10.1109/MICRO.2003.1253181.

[Muk+03b] S.S. Mukherjee, C.T. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. "Measuring architectural vulnerability factors." In: *IEEE Micro* 23.6 (2003), pp. 70–75. DOI: 10.1109/MM.2003.1261389.

[Muk08] Shubu Mukherjee. *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 978-0-12-369529-1.

[NX06] V. Narayanan and Y. Xie. "Reliability concerns in embedded system designs." In: *Computer* 39.1 (2006), pp. 118–120. DOI: 10.1109/MC.2006.31.

[Nic10] M. Nicolaidis. *Soft Errors in Modern Electronic Systems*. Frontiers in Electronic Testing. Springer US, 2010. ISBN: 9781441969934.

[Nor96a] E. Normand. "Single event upset at ground level." In: *IEEE Transactions on Nuclear Science* 43.6 (Dec. 1996), pp. 2742–2750. ISSN: 0018-9499. DOI: 10.1109/23.556861.

[Nor96b] E. Normand. "Single event upset at ground level." In: *IEEE Transactions on Nuclear Science* 43.6 (1996), pp. 2742–2750. DOI: 10.1109/23.556861.

[Nul98] Frank Nullmeier. "Input, Output, Outcome, Effektivität und Effizienz." In: *Handbuch zur Verwaltungsreform*. Wiesbaden: VS Verlag für Sozialwissenschaften, 1998, pp. 314–322. ISBN: 978-3-322-83673-1. DOI: 10.1007/978-3-322-83673-1_35. URL: https://doi.org/10.1007/978-3-322-83673-1_35.

[OR95] J. Ohlsson and M. Rimen. "Implicit signature checking." In: *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS '95))*. Vol. 25. 1995, pp. 218–227. DOI: 10.1109/FTCS.1995.466976.

[OSM02] N. Oh, P.P. Shirvani, and E.J. McCluskey. "Error detection by duplicated instructions in super-scalar processors." In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 63–75. ISSN: 0018-9529. DOI: 10.1109/24.994913.

[Ose] *OSEK/VDX Group Homepage*. http://www.osek-vdx.org/.

[PG21] George Papadimitriou and Dimitris Gizopoulos. "Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers." In: *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. 2021, pp. 902–915. DOI: 10.1109/ISCA52012.2021.00075.

[PGZ08] Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn. "Samurai: Protecting Critical Data in Unsafe Languages." In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)*. Eurosys '08. Glasgow, Scotland UK: Association for Computing Machinery, 2008, pp. 219–232. ISBN: 9781605580135. DOI: 10.1145/1352592.1352616. URL: https://doi.org/10.1145/1352592.1352616.

[PH21] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 6th ed. Morgan Kaufmann Publishers Inc., 2021. ISBN: 978-0-12-823716-8.

[PRSR98] P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. "Exploiting the background debugging mode in a fault injection system." In: *Proceedings. IEEE International Computer Performance and Dependability Symposium. IPDS'98 (Cat. No.98TB100248)*. 1998, pp. 277–. DOI: 10.1109/IPDS.1998.707736.

[Pal+19] 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)*. 2019, pp. 151–162. DOI: 10.1109/ISSRE.2019.00024.

[Par+00a] B. Parrotta, M. Rebaudengo, M. Sonza Reorda, and M. Violante. "Speeding-Up Fault Injection Campaigns in VHDL Models." In: *Computer Safety, Reliability and Security*. Ed. by Floor Koornneef and Meine van der Meulen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 27–36. ISBN: 978-3-540-40891-8. DOI: 10.1007/3-540-40891-6_3.

[Par+00b] B. Parrotta, M. Rebaudengo, M.S. Reorda, and M. Violante. "New techniques for accelerating fault injection in VHDL descriptions." In: *Proceedings 6th IEEE International On-Line Testing Workshop (Cat. No. PR00646)*. IEEE. IEEE Computer Society Press, 2000, pp. 61–66. DOI: 10.1109/OLT.2000.856613.

[Par+14] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D. Antonopoulos, and Nikolaos Bellas. "GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates." In: *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 2014, pp. 622–629. DOI: 10.1109/DSN.2014.96.

[Par72] David Lorge Parnas. "On the Criteria to be used in Decomposing Systems into Modules." In: *Communications of the ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623.

[Pau04] Paul. "Electronic Elections in Belgium: a look at the sourcecode." In: *Afront.be* (2004). URL: https://web.archive.org/web/20080107081704/http://www.afront.be/lib/vote.html (visited on 2024-03-28).

[Pro22] Community Project. *gem5 Computer Architecture Research Project*. 2022. URL: https://gem5.googlesource.com/public/gem5 (visited on 2024-03-28).

[Pro23a] Community Project. *Bochs IA-32 Emulator Project*. 2023. URL: https://github.com/bochs-emu/Bochs (visited on 2024-03-28).

[Pro23b] Research Project. *Sail*. 2023. URL: https://www.cl.cam.ac.uk/~pes20/sail/ (visited on 2024-03-28).

[RBQ96] C. Rabejac, J.-P. Blanquart, and J.-P. Queille. "Executable assertions and timed traces for on-line software error detection (FTCS '96)." In: *Proceedings of Annual Symposium on Fault Tolerant Computing*. 1996, pp. 138–147. DOI: 10.1109/FTCS.1996.534602.

[RCE02] M. Redeker, B.F. Cockburn, and D.G. Elliott. "An investigation into crosstalk noise in DRAM structures." In: *Proceedings of the 2002 IEEE International Workshop on Memory Technology, Design and Testing (MTDT2002)*. 2002, pp. 123–129. DOI: 10.1109/MTDT.2002.1029773.

[RGF23] Federico Reghenzani, Zhishan Guo, and William Fornaciari. "Software Fault Tolerance in Real-Time Systems: Identifying the Future Research Questions." In: *ACM Comput. Surv.* 55.14s (2023), pp. 1–30. ISSN: 0360-0300. DOI: 10.1145/3589950.

[RM00] Steven K. Reinhardt and Shubhendu S. Mukherjee. "Transient Fault Detection via Simultaneous Multithreading." In: *Proceedings of the 27th International Symposium on Computer Architecture (ISCA '00)*. Vancouver, British Columbia, Canada: ACM Press, 2000, pp. 25–36. ISBN: 1-58113-232-8. DOI: 10.1145/339647.339652.

[RSR99] M. Rebaudengo and M. Sonza Reorda. “Evaluating the fault tolerance capabilities of embedded systems via BDM.” In: *Proceedings 17th IEEE VLSI Test Symposium (Cat. No.PR00146)*. 1999, pp. 452–457. DOI: 10.1109/VTEST.1999.766703.

[Ram+08] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda. “Statistical Fault Injection.” In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. June 2008, pp. 122–127. DOI: 10.1109/DSN.2008.4630080.

[Ram21] (pseudonym) Ramble_Khron. “An Eight Year Mystery Solved After Speedrunner Gets Help From Space.” In: *ebaumsworld.com* (Sept. 15, 2021). URL: https://gaming.ebaumsworld.com/articles/how-cosmic-rays-helped-a-mario-speedrunner-do-the-impossible/86984570/ (visited on 2024-03-28).

[Rat04] David Ratter. “FPGAs on mars.” In: *Xcell J* 50 (2004), pp. 8–11. URL: https://www.xilinx.com/publications/archives/xcell/Xcell50.pdf (visited on 2024-03-28).

[Rei+05a] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. “SWIFT: software implemented fault tolerance.” In: *International Symposium on Code Generation and Optimization (CGO ’05)*. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.

[Rei+05b] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, D.I. August, and S.S. Mukherjee. “Software-controlled fault tolerance.” In: *ACM Transactions on Architecture and Code Optimization (TACO ’05)* 2.4 (2005), pp. 366–396. ISSN: 1544-3566. DOI: 10.1145/1113841.1113843.

[Ros+11] Daniele Rossi, Nicola Timoncini, Michael Spica, and Cecilia Metra. “Error correcting code analysis for cache memory high reliability and performance.” In: *2011 Design, Automation & Test in Europe*. IEEE. 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763257.

[Rot99] E. Rotenberg. “AR-SMT: a microarchitectural approach to fault tolerance in microprocessors.” In: *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (FTCS ’99)*. Vol. 29. Washington, DC, USA: IEEE Computer Society Press, 1999, pp. 84–91. DOI: 10.1109/FTCS.1999.781037.

[SB19] Horst Schirmeier and Mark Breddemann. “Quantitative Cross-Layer Evaluation of Transient-Fault Injection Techniques for Algorithm Comparison.” In: *15th European Dependable Computing Conference, EDCC 2019, Naples, Italy, September 17-20, 2019* (Naples, Italy). Piscataway, NJ, USA: IEEE Press, Sept. 2019, pp. 15–22. DOI: 10.1109/EDCC.2019.00016.

[SBK10] D. Skarin, R. Barbosa, and J. Karlsson. “GOOFI-2: A tool for experimental dependability assessment.” In: *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN ’09)*. IEEE Computer Society Press, June 2010, pp. 557–562. DOI: 10.1109/DSN.2010.5544265.

[SBS14] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. “Rapid Fault-Space Exploration by Evolutionary Pruning.” In: *International Conference on Computer Safety, Reliability, and Security*. Ed. by Andrea Bondavalli and Felicita Di Giandomenico. Cham: Springer International Publishing, 2014, pp. 17–32. ISBN: 978-3-319-10506-2.

[SH+14] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V. Adve, and Helia Naeimi. “GangES: Gang Error Simulation for Hardware Resiliency Evaluation.” In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 61–72. ISSN: 0163-5964. DOI: 10.1145/2678373.2665685. URL: https://doi.org/10.1145/2678373.2665685.

[SJK17] Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. "Light-Weight Techniques for Improving the Controllability and Efficiency of ISA-Level Fault Injection Tools." In: *Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on*. IEEE. 2017, pp. 68–77. DOI: 10.1109/PRDC.2017.18.

[SK08a] Daniel Skarin and Johan Karlsson. "Software Implemented Detection and Recovery of Soft Errors in a Brake-by-Wire System." In: *2008 Seventh European Dependable Computing Conference (EDCC '08)*. 2008, pp. 145–154. DOI: 10.1109/EDCC-7.2008.24.

[SK08b] Vilas Sridharan and David R. Kaeli. "Quantifying software vulnerability." In: *WREFT '08: Proceedings of the 2008 workshop on Radiation effects and fault tolerance in nanometer technologies* (Ischia, Italy). New York, NY, USA: ACM Press, 2008, pp. 323–328. ISBN: 978-1-60558-092-9. DOI: 10.1145/1366224.1366225.

[SK09] V. Sridharan and D. R. Kaeli. "Eliminating microarchitectural dependency from Architectural Vulnerability." In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. Feb. 2009, pp. 117–128. DOI: 10.1109/HPCA.2009.4798243.

[SL12] Vilas Sridharan and Dean Liberty. "A study of DRAM failures in the field." In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–11. DOI: 10.1109/SC.2012.13.

[SM98] Philip P. Shirvani and Edward J. McCluskey. *Fault-Tolerant Systems in a Space Environment: The CRC ARGOS project*. Tech. rep. 2. Stanford, CA, USA, 1998.

[SRS14] Horst Schirmeier, Lars Rademacher, and Olaf Spinczyk. "Smart-Hopping: Highly Efficient ISA-Level Fault Injection on Real Hardware." In: *Proceedings of the 19th IEEE European Test Symposium (ETS '14)*. Paderborn, Germany: IEEE Computer Society Press, May 2014. DOI: 10.1109/ETS.2014.6847803.

[SSM00] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey. "Software-implemented EDAC protection against SEUs." In: *IEEE Transactions on Reliability* 49.3 (Sept. 2000), pp. 273–284. ISSN: 0018-9529. DOI: 10.1109/24.914544.

[STB97] Volkmar Sieh, Oliver Tschäche, and Frank Balbach. "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions." In: *Proceedings of the 27rd International Symposium on Fault-Tolerant Computing (FTCS-27)*. June 1997, pp. 32–36. DOI: 10.1109/FTCS.1997.614074.

[SWP13] Jiguo Song, J. Wittrock, and G. Parmer. "Predictable, Efficient System-Level Fault Tolerance in C^3." In: *Proceedings of the 34th IEEE International Symposium on Real-Time Systems (RTSS '13)* (Vancouver, Canada, Dec. 3–6, 2013). IEEE Computer Society Press, Dec. 2013, pp. 21–32. ISBN: 978-1-4799-2007-5. DOI: 10.1109/RTSS.2013.11.

[Sai77] S.H. Saib. "Executable Assertions - An Aid To Reliable Software." In: *1977 11th Asilomar Conference on Circuits, Systems and Computers, 1977. Conference Record.* 1977, pp. 277–281. DOI: 10.1109/ACSSC.1977.748932.

[San+14] T. Santini, P. Rech, G. Nazar, L. Carro, and F. Rech Wagner. "Reducing embedded software radiation-induced failures through cache memories." In: *Proceedings of the 19th IEEE European Test Symposium (ETS '14)*. Paderborn, Germany: IEEE Computer Society Press, May 2014, pp. 1–6. DOI: 10.1109/ETS.2014.6847793.

[Sch+10a] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. "ANB- and ANBDmem-encoding: detecting hardware errors in software." In: *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '10)* (Vienna, Austria). Ed. by Erwin Schoitsch. Heidelberg, Germany: Springer-Verlag, 2010, pp. 169–182. ISBN: 978-3-642-15650-2. DOI: 10.1007/978-3-642-15651-9_13.

[Sch+10b] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. "Slice Your Bug: Debugging Error Detection Mechanisms Using Error Injection Slicing." In: *2010 European Dependable Computing Conference*. 2010, pp. 13–22. DOI: 10.1109/EDCC.2010.12.

[Sch11] Ute Schiffel. "Hardware Error Detection Using AN-Codes." PhD thesis. Technische Universität Dresden, Fakultät Informatik, 2011. URL: https://tud.qucosa.de/api/qucosa%3A25596/attachment/ATT-0/?L=1 (visited on 2024-03-28).

[Sch+13] Bernard Schmidt, Carlos Villarraga, Thomas Fehmel, Jörg Bormann, Markus Wedler, Minh Nguyen, Dominik Stoffel, and Wolfgang Kunz. "A New Formal Verification Approach for Hardware-Dependent Embedded System Software." In: *IPSJ Transactions on System LSI Design Methodology* 6.0 (2013), pp. 135–145. ISSN: 1882-6687. DOI: 10.2197/ipsjtsldm.6.135.

[Sch+15] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance." In: *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*. Ed. by Pierre Sens. Paris, France, Sept. 2015, pp. 245–255. DOI: 10.1109/EDCC.2015.28.

[Sch16] Horst Schirmeier. "Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance." Dissertation. Technische Universität Dortmund, July 2016. DOI: 10.17877/DE290R-17222. URL: https://eldorado.tu-dortmund.de/bitstream/2003/35175/1/Dissertation.pdf (visited on 2024-03-29).

[Sch97] R.R. Schaller. "Moore's law: past, present and future." In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665.

[Shi+00] Philip P Shirvani, Namsuk Oh, Edward J Mccluskey, DL Wood, Michael N Lovellette, and KS Wood. "Software-implemented hardware fault tolerance experiments: COTS in space." In: *Processings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '00)*. IEEE Computer Society Press, 2000, B56–B57.

[Shi+02] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic." In: *Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN '02)* (Bethesda, MD, USA). Washington, DC, USA: IEEE Computer Society Press, June 2002, pp. 389–398. DOI: 10.1109/DSN.2002.1028924.

[Shy+07] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance." In: *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN '07)* (Edinburgh, UK). Washington, DC, USA: IEEE Computer Society Press, June 2007, pp. 297–306. ISBN: 0-7695-2855-4. DOI: 10.1109/DSN.2007.98.

[Shy+09] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures." In: *IEEE Transactions on Dependable and Secure Computing* 6.2 (2009), pp. 135–148. DOI: 10.1109/TDSC.2008.62.

[Sle+99] T. J. Slegel, R. M. Averill, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. "IBM's S/390 G5 microprocessor design." In: *IEEE Micro* 19.2 (Mar. 1999), pp. 12–23. ISSN: 0272-1732. DOI: 10.1109/40.755464.

[Smi+95] D Todd Smith, Barry W Johnson, Joseph A Profeta, and Daniele G Bozzolo. "A method to determine equivalent fault classes for permanent and transient faults." In: *Reliability and Maintainability Symposium, 1995. Proceedings., Annual*. IEEE. 1995, pp. 418–424. DOI: 10.1109/RAMS.1995.513278.

[Sri+04] Jayanth Srinivasan, Sarita V Adve, Pradip Bose, and Jude A Rivers. "The impact of technology scaling on lifetime reliability." In: *Proceedings of the 34th International Conference on Dependable Systems and Networks (DSN '04)* (Florence, Italy). Washington, DC, USA: IEEE Computer Society Press, June 2004, pp. 177–186. DOI: 10.1109/DSN.2004.1311888.

[Sri+13] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. "Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults." In: *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM Press, 2013, 22:1–22:11. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503257.

[Sri+15] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly." In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Istanbul, Turkey: ACM, 2015. DOI: 10.1145/2694344.2694348.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. "Simultaneous multithreading: Maximizing on-chip parallelism." In: *Proceedings of the 22nd annual international symposium on Computer architecture (ISCA'95)*. 1995, pp. 392–403. DOI: 10.1145/223982.224449.

[TI95] Timothy K. Tsai and Ravishankar K. Iyer. "Measuring Fault Tolerance with the FTAPE fault injection tool." In: *Quantitative Evaluation of Computing and Communication Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 26–40. ISBN: 978-3-540-44789-4.

[TN93] A. Taber and E. Normand. "Single event upset in avionics." In: *IEEE Transactions on Nuclear Science* 40.2 (Apr. 1993), pp. 120–126. ISSN: 0018-9499. DOI: 10.1109/23.212327.

[TP13] Anna Thomas and Karthik Pattabiraman. "LLFI : An Intermediate Code Level Fault Injector For Soft Computing Applications." In: *Workshop on Silicon Errors in Logic System Effects (SELSE)*. 2013. URL: https://api.semanticscholar.org/CorpusID:202726208 (visited on 2024-03-28).

[Tri08] Kishor S Trivedi. *Probability & statistics with reliability, queuing and computer science applications*. John Wiley & Sons, 2008. ISBN: 9780471333418.

[Tur37] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem.” In: *Proceedings of the London Mathematical Society*. Proceedings of the London Mathematical Society s2-42.1 (1937), pp. 230–265. ISSN: 0024-6115, 1460-244X. DOI: 10.1112/plms/s2-42.1.230.

[Tyr96] A.M. Tyrrell. “Recovery blocks and algorithm-based fault tolerance.” In: *Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies*. Vol. 22. IEEE Computer Society Press, 1996, pp. 292–299. DOI: 10.1109/EURMIC.1996.546394.

[Ulb+12] Peter Ulbrich, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Reiner Schmid. “Eliminating Single Points of Failure in Software-Based Redundancy.” In: *Proceedings of the 9th European Dependable Computing Conference (EDCC ’12)* (Sibiu, Romania). Washington, DC, USA: IEEE Computer Society Press, May 2012, pp. 49–60. ISBN: 978-1-4673-0938-7. DOI: 10.1109/EDCC.2012.21.

[Ulb14] Peter Ulbrich. “Ganzheitliche Fehlertoleranz in eingebetteten Softwaresystemen.” PhD thesis. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg, 2014.

[VPC02] T.N. Vijaykumar, I. Pomeranz, and K. Cheng. “Transient-fault recovery using simultaneous multithreading.” In: *Proceedings 29th Annual International Symposium on Computer Architecture (ISCA ’02)*. Vol. 29. IEEE Computer Society Press, 2002, pp. 87–98. DOI: 10.1109/ISCA.2002.1003565.

[Val+15] A. Vallero, S. Tselonis, N. Foutris, M. Kaliorakis, M. Kooli, A. Savino, G. Politano, A. Bosio, G. Di Natale, D. Gizopoulos, and S. Di Carlo. “Cross-layer Reliability Evaluation, Moving from the Hardware Architecture to the System Level.” In: *Microprocess. Microsyst.* 39.8 (Nov. 2015), pp. 1204–1214. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2015.06.003.

[Val21] [@FiloSottile] Filippo Valsorda. *A cosmic ray just murdered a Certificate Transparency log.* X (formerly known as Twitter). July 4, 2021. URL: https://x.com/FiloSottile/status/1411583960115814401 (visited on 2024-03-28).

[Vel+99] R. Velazco, P. Cheynet, A. Tissot, J. Haussy, J. Lambert, and R. Ecoffet. “Evidences of SEU tolerance for digital implementations of artificial neural networks: one year MPTB flight results.” In: *1999 5th European Conference on Radiation and Its Effects on Components and Systems (RADECS ’99)*. Vol. 5. IEEE Computer Society Press, 1999, pp. 565–568. DOI: 10.1109/RADECS.1999.858648.

[Ven+16] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. “Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency.” In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–14. DOI: 10.1109/MICRO.2016.7783745.

[Ven+19] R. Venkatagiri, K. Ahmed, A. Mahmoud, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve. “gem5-Approxilyzer: An Open-Source Tool for Application-Level Soft Error Analysis.” In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019, pp. 214–221. DOI: 10.1109/DSN.2019.00033.

[WA19] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: Unpriviledged ISA – Document Version 20191213*. Dec. 2019. URL: https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view?usp=drive_link (visited on 2024-03-28).

[WAH21] Andrew Waterman, Krste Asanovic, and John Hauser, eds. *The RISC-V Instruction Set Manual – Volume II: Privileged Architecture, Document Version 20211203"*. Dec. 2021. URL: https://drive.google.com/file/d/1EMip5dZlnypTk7pt4WWUKmtjUKTOkBqh/view?usp=drive_link (visited on 2024-03-28).

[WF07] Ute Wappler and Christof Fetzer. "Software Encoded Processing: Building Dependable Systems with Commodity Hardware." In: *Proceedings of the 26th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '07)* (Nuremberg, Germany). Ed. by Francesca Saglietti and Norbert Oster. Heidelberg, Germany: Springer-Verlag, 2007, pp. 356–369. ISBN: 978-3-540-75100-7. DOI: 10.1007/978-3-540-75101-4_34.

[WH11] N.H.E. Weste and D.M. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2011. ISBN: 9780321547743.

[Wei84] Reinhold P Weicker. "Dhrystone: a synthetic systems programming benchmark." In: *Communications of the ACM* 27.10 (1984), pp. 1013–1030. DOI: 10.1145/358274.358283.

[Wil02a] Dennis J Wilkins. "The Bathtub Curve and Product Failure Behavior Part One - The Bathtub Curve, Infant Mortality and Burn-in." In: *Reliability HotWire* 21 (2002). URL: https://www.maths.tcd.ie/~donmoore/project/project/Write%20up/22%20mar%202006/hottopics21.htm (visited on 2024-03-28).

[Wil02b] Dennis J Wilkins. "The Bathtub Curve and Product Failure Behavior Part Two - Normal Life and Wear-Out." In: *Reliability HotWire* 22 (2002). URL: https://www.maths.tcd.ie/~donmoore/project/project/Write%20up/22%20mar%202006/hottopics22.htm (visited on 2024-03-28).

[Win+13] 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)*. 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575310.

[Win+15] Stefan Winter, Thorsten Piper, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. "GRINDER: On Reusability of Fault Injection Tools." In: *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*. 2015, pp. 75–79. DOI: 10.1109/AST.2015.22.

[XL12] Xin Xu and Man-Lap Li. "Understanding soft error propagation using efficient vulnerability-driven fault injection." In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE. 2012, pp. 1–12. DOI: 10.1109/DSN.2012.6263923.

[XTS08] Jianjun Xu, Qingping Tan, and Rui Shen. "A Novel Optimum Data Duplication Approach for Soft Error Detection." In: *2008 15th Asia-Pacific Software Engineering Conference (APSEC '08)*. 2008, pp. 161–168. DOI: 10.1109/APSEC.2008.46.

[YC80] S.S. Yau and Fu-Chung Chen. “An Approach to Concurrent Control Flow Checking.” In: *IEEE Transactions on Software Engineering* SE-6.2 (Mar. 1980), pp. 126–137. ISSN: 0098-5589. DOI: 10.1109/TSE.1980.234478.

[YE10] Doe Hyun Yoon and Mattan Erez. “Virtualized and flexible ECC for main memory.” In: *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2010, pp. 397–408. ISBN: 9781605588391. DOI: 10.1145/1736020.1736064.

[Yeh96] Y.C. Yeh. “Triple-triple redundant 777 primary flight computer.” In: *Proceedings of the 1996 IEEE Aerospace Applications Conference* (Aspen, CO, USA). Washington, DC, USA: IEEE Computer Society Press, Feb. 1996, pp. 293–307. ISBN: 978-0780331969. DOI: 10.1109/AERO.1996.495891.

[Yos13] Junko Yoshida. *Toyota Case: Single Bit Flip That Killed*. 2013. URL: https://www.eetimes.com/toyota-case-single-bit-flip-that-killed/ (visited on 2024-03-28).

[ZL79] J. F. Ziegler and W. A. Lanford. “Effect of Cosmic Rays on Computer Memories.” In: *Science* 206.4420 (1979), pp. 776–788. DOI: 10.1126/science.206.4420.776. URL: https://www.science.org/doi/abs/10.1126/science.206.4420.776 (visited on 2024-03-28).

[Zie+96] J.F. Ziegler, H.W. Curtis, H.P. Muhlfeld, C.J. Montrose, B. Chin, M. Nicewicz, C.A. Russell, W.Y. Wang, L.B. Freeman, P. Hosier, L.E. LaFave, J.L. Walsh, J.M. Orro, G.J. Unger, J.M. Ross, T.J. O’Gorman, B. Messina, T.D. Sullivan, A.J. Sykes, H. Yourke, T.A. Enger, V. Tolat, T.S. Scott, A.H. Taber, R.J. Sussman, W.A. Klein, and C.W. Wahaus. “IBM experiments in soft fails in computer electronics (1978–1994).” In: *IBM Journal of Research and Development* 40.1 (Jan. 1996), pp. 3–18. ISSN: 0018-8646. DOI: 10.1147/rd.401.0003.

List of Figures

List of Tables

List of Listings