Technische Universität München

Lehrstuhl für Sicherheit in der Informationstechnik
an der Fakultät für Elektrotechnik und Informationstechnik

New Techniques for Emulating Fault Attacks

Ralph Heinz-Erik Nyberg

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines **Doktor-Ingenieurs (Dr.-Ing.)** genehmigten Dissertation.

Vorsitzender:	apl. Prof. Dr.-Ing. Walter Stechele
Prüfer der Dissertation:	1. Prof. Dr.-Ing. Georg Sigl
	2. Prof. Dr.-Ing. Ulf Schlichtmann
	3. Prof. Dr.-Ing. Dirk Rabe

Die Dissertation wurde am 17.04.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 04.09.2018 angenommen.

Abstract

Integrated security circuits such as smart cards and security controllers protect sensitive data and protect its integrity with dedicated measures, which are often based on secured implementations of cryptographic algorithms. Besides the typical areas, identification, pay TV and ticketing, integrated security circuits are also deployed in automotive and industrial applications. Usually, countermeasures against so-called fault attacks are implemented in software and hardware to prevent attackers from compromising sensible data with fault attacks. These countermeasures have to be verified already during circuit design taking the specified security requirements into account, which is also known as security verification. For security verification of digital fault countermeasures, pre-silicon fault injection techniques are usually used to model complex physical effects of fault attacks during circuit design. Although abstract fault models are used, the huge variety of fault parameters in terms of spatial and temporal fault injection lead to a fault injection space infeasible to be analyzed completely. Therefore, analysis need to focus on subsets of the fault injection space, which have to be selected based on their relevance for the specified security level. These relevant faults have to be modeled with efficient tools that, on the one hand, provide the appropriate configurability and, on the other hand, provide high-performance to make security verification feasible in given time.

FPGA-based fault emulation, which basically constitutes a hardware accelerated fault simulation technique where the circuit under verification is synthesized onto an FPGA, features the highest possible pre-silicon verification performance. While a test sensitizes the circuit under verification, configurable faults are injected into it by additional hardware components synthesized onto the FPGA along with the actual circuit to be verified. Unfortunately, depending on the implementation, state-of-the-art fault emulation lacks either the configurability required to model arbitrary fault attacks or the performance is limited by a communication bottleneck between software and hardware components for fault configuration. The goal of my thesis is to close the gap between configurability and performance of FPGA-based fault emulation and to achieve industrial applicability for security related designs.

First, I introduce a meta fault configuration model able to describe arbitrary fault configurations, which I use afterwards to describe the features of the implemented fault emulation techniques. The fault configuration model is defined at a meta level to make it independent of utilized tools, the circuits to be verified and their level of abstraction to be analyzed. This enables the description of fault models known from literature as well as new ones, and reuse for other applications is enabled.

Next, I present an FPGA-based fault emulation environment that supports the configurability required to mimic arbitrary fault attacks in combinational and sequential logic. My implementation supports fault configuration at runtime, which includes configurable fault model types, single and multiple faults and variable fault durations. Fault configurations can be applied at arbitrary discrete times during fault experiments. To allow the injection of time-displaced or time-overlapped faults, I propose a feature to reconfigure the fault settings during runtime in an efficient way, requiring a negligible amount of additional logic. This enables the emulation of sophisticated fault attacks such as exploiting multiple laser beams to inject faults with independent sources in a time-displaced or a time-overlapped fashion.

Further, I propose a hardware-efficient and performance-efficient concept for evaluating emulation results in hardware. This method is supported by a self-testing functional test software for processor-based security designs. A single fault-free emulation is used to generate a golden reference for following fault emulations. This way, the reference for result comparison has not to be generated by a second instance of the circuit to be verified. Moreover, the fault emulator is sensitive to events on dedicated observation points, which allows to judge the circuit behavior in a more flexible manner and reduces false positives in the emulation results. This feature also allows to configure temporal fault injection settings relatively to dedicated events on the defined observation points. For instance, time frames of varying length in which security-relevant operations are executed can be selected for fault injection without manually determining the absolute timing of them.

There are usually considerably more combinational cells than sequential cells in the designs to be verified. As a consequence, instrumenting circuits for fault injection in combinational logic consumes more of the limited hardware resources on the FPGA. Unfortunately, the entire combinational logic needs to be instrumented to support arbitrary multiple fault injection at runtime, which limits applicability compared to fault injection in sequential logic. Moreover, the performance is decreased since timing paths are increased with each instrumented combinational cell. To address these limitations, I present a software-based pre-processor that maps faults in combinational logic to equivalent faults in sequential logic. These equivalent faults are then configured into the fault emulator for performance efficient analysis.

Once I achieved my goals w.r.t. configurability and applicability for industrial designs, I focus on performance optimization measures, which finally close the gap between performance and configurability. The presented optimizations address the communication bottleneck of FPGA-based fault emulation, reduce emulation runtime and cancel fault experiments when these exhibit fault effects equivalent to previous experiments.

To discuss the improvements of my work over the state-of-the-art, I applied the presented fault emulation techniques to two different microcontroller designs and configured hundreds of millions of fault attacks. With my results I demonstrate that the presented techniques enable to model arbitrary fault attacks in combinational and sequential logic of industrial designs without performance loss compared to fault-free emulations. As far as fault injection in combinational logic is concerned, I removed the limitation on applicability, I reduced the hardware requirements by 45% and I increased the performance by a factor of six at the same time. The proposed techniques allow to mimic more arbitrary fault attacks in given time and thus help to improve security devices by means of possibly finding more security flaws already during circuit design, i.e. before the manufactured devices are analyzed during post-silicon certification.

Kurzfassung

Integrierte Sicherheits-Schaltungen – im Sinne des englischen Wortes *security* – schützen sensible Daten und ihre Integrität. Hierzu werden Sicherheitsmaßnahmen implementiert, die häufig auf abgesicherten Implementierungen kryptographischer Verfahren basieren. Zu integrierten Sicherheits-Schaltungen gehören Chipkarten und Sicherheitskontroller, die zum Beispiel bei elektronischen Reisepässen, Zugangskarten, elektrischen Tickets für öffentliche Verkehrsmittel, sowie für Automotive- und Industrieanwendungen eingesetzt werden. Um eine Kompromittierung sensibler Daten durch sogenannte Fehlerattacken auf integrierte Schaltungen zu verhindern, werden gewöhnlich dedizierter Gegenmaßnahmen in Hard- und Software implementiert. Während der Entwicklung von integrierten Sicherheits-Schaltungen müssen die implementierten Sicherheitsmaßnahmen unter Berücksichtigung der spezifizierten Sicherheitsanforderungen verifiziert werden. Zur Sicherheitsverifikation von digitalen Gegenmaßnahmen hinsichtlich Fehlerattacken werden hierzu üblicherweise Fehlerinjektionstechniken eingesetzt, welche die komplexen physikalischen Auswirkungen von Fehlerattacken bereits während der Entwicklung des Schaltungsdesigns modellieren. Trotz abstrakter Modelle ist es unmöglich, alle möglichen Fehlerattacken zu modellieren, da berücksichtigt werden muss, dass ein Angreifer örtliche und zeitliche Angriffspunkte beliebig variieren kann, was zu einem komplexen Fehlerraum führt. Daher müssen für das zu erzielende Schutzniveau relevante Submengen des Fehlerraums selektiert, anschließend modelliert und basierend auf der Modellierung analysiert werden. Dies erfordert Modellierungswerkzeuge, die zum einen die Konfiguration entsprechender Fehlerattacken ermöglichen und zum anderen sehr performant sind, um die Menge der analysierbaren Fehlerattacken in der verfügbaren Zeit für die Sicherheitsverifikation zu maximieren.

Die FPGA-basierte Fehleremulation, welche prinzipiell eine Hardware-beschleunigte Fehlersimulation darstellt, ist das schnellste Werkzeug zur Modellierung von Fehlern. Hierzu wird die zu verifizierende Schaltung in ein FPGA konfiguriert. Während die Schaltung durch einen Test sensitiviert wird, werden konfigurierbare Fehler mittels zusätzlicher Hardware, die eben-

falls in das FPGA konfiguriert ist, in die zu verifizierende Schaltung injiziert.

Im Rahmen dieser Arbeit schließe ich mittels neuer Emulationstechniken die Lücke zwischen Konfigurierbarkeit und Geschwindigkeit der FPGA-basierten Fehleremulation und ermögliche die Anwendbarkeit für industrielle Sicherheitskontroller. Die präsentierten Emulationstechniken erlauben es beliebige Fehler zu konfigurieren, was in dieser Form mit bisherigen Ansätzen entweder nicht möglich ist, oder aufgrund eines Kommunikationsflaschenhalses zu erheblichen Geschwindigkeitsverlusten führt.

Zuerst führe ich ein Meta-Fehlerkonfigurationsmodell ein, welches ich verwende, um die Konfigurationsmöglichkeiten der implementierten Fehlerinjektionstechniken eindeutig zu beschreiben. Das Meta-Fehlerkonfigurationsmodell ist auf einem abstrakten Meta-Level verallgemeinert beschrieben, sodass es unabhängig von den eingesetzten Modellierungswerkzeugen, der zu verifizierenden Schaltung und dem zur Verifikation betrachteten Abstraktionslevels der Schaltung ist. So lassen sich bekannte Fehlermodelle aus der Literatur sowie neue Fehlermodelle beschreiben und die Wiederverwendbarkeit für andere Applikationen ist sichergestellt.

Anschließend präsentiere ich eine FPGA-basierte Fehleremulationsumgebung, die die erforderliche Konfigurierbarkeit zur Modellierung beliebiger Fehlerattacken in kombinatorischer und sequentieller Logik bietet. Meine Implementierung unterstützt zur Laufzeit konfigurierbare Fehlermodelle sowie Einzel- und Mehrfachfehler mit einer variablen Injektionsdauer. Eine Fehlerkonfiguration kann einmal pro Fehlerexperiment zu beliebigen diskreten Zeitpunkten aktiviert werden. Ich präsentiere ein zusätzliches Feature, welches die Aktivierung verschiedener Fehlerkonfigurationen zu mehreren beliebigen diskreten Zeitpunkten während eines Fehlerexperiments ermöglicht. So lassen sich aufwändige Fehlerattacken konfigurieren, die zum Beispiel mit mehreren Laserstrahlen von unabhängigen Quellen zeitlich überlagert oder zeitversetzt durchgeführt werden können.

Des Weiteren schlage ich ein Konzept zur Evaluation der Emulationsergebnisse in Hardware vor. Für Prozessor-basierte Schaltungsdesigns wird dieser Ansatz durch die Verwendung eines selbsttenstenden funktionalen Tests unterstützt. Mit Hilfe einer einzigen fehlerfreien Emulation wird eine Referenz für alle folgende Fehleremulationen generiert, wodurch die Referenz nicht durch eine Duplikation des zu verifizierenden Designs generiert werden muss. Dieser Ansatz ist effizient hinsichtlich des benötigten Hardwareaufwands sowie der erzielten Geschwindigkeit. Beobachtungspunkte, an welchen die Emulation auf bestimmte Ereignisse sensitiv ist, erlauben eine flexiblere Bewertung des Verhaltens des zu verifizierenden Schaltungsdesigns, wodurch falsch-positive Emulationsergebnisse reduziert werden. Außerdem können Zeitpunkte für die Fehlerinjektion relativ zu dedizierten Ereignissen an den definierten Beobachtungspunkten konfiguriert werden. Dadurch wird es ermöglicht, Zeitbereiche variabler Länge für die Fehlerinjektion zu berücksichtigen, in denen sicherheitsrelevante Operationen durchgeführt werden, ohne die absoluten Zeitpunkte manuell zu bestimmen.

Die zu verifizierenden Schaltungen können als synchrone sequentielle Schaltung realisiert sein, welche aus kombinatorischen und sequentiellen Gattern bestehen, wobei der Anteil an kombinatorischen Gattern üblicherweise wesentlich höher ist. Um die Modellierung beliebiger Fehler innerhalb eines Schaltungsdesigns zu unterstützen, muss jedes Gatter für eine Fehlerinjektion instrumentiert werden. Für jedes Gatter wird zusätzliche Logik zur Steuerung der Fehlerinjektion benötigt, wodurch der Hardwareaufwand zur Realisierung der Fehlerinjektion entsprechend der Anzahl an kombinatorischen und sequentiellen Gattern des zu verifizierenden Schaltungsdesigns steigt. Verglichen mit der FPGA-basierten Fehleremulation für Fehler in ausschließlich sequentieller Logik, wird daher für die Fehlerinjektion in ausschließlich kombinatorischer Logik mehr der verfügbaren Hardware des verwendeten FPGAs benötigt. Dies hat zur Konsequenz, dass die maximale Schaltungsgröße des zu verifizierenden Schaltungsdesigns kleiner ist, wenn die FPGA-basierte Fehleremulation für Fehler in kombinatorischer Logik realisiert wird. Um die daraus resultierende Limitierung der Anwendbarkeit der FPGA-basierten Fehleremulation für die Fehlerinjektion in kombinatorischer Logik zu beseitigen, präsentiere ich einen Software-basierten Präprozessor, welcher Fehler in kombinatorischer Logik auf äquivalente Fehler in sequentieller Logik abbildet, die dann performant mit dem Fehleremulator untersucht werden können. Dieser Ansatz bietet weitere Vorteile, da äquivalente Fehler bestimmt und zur Geschwindigkeitsoptimierung von weiteren Untersuchungen ausgeschlossen werden können.

Da mittels der präsentierten Fehleremulationstechniken meine Ziele hinsichtlich Konfigurierbarkeit und Anwendbarkeit erfüllt sind, widme ich mich als nächstes Geschwindigkeitsoptimierungen, um die Lücke zwischen Konfigurierbarkeit und Geschwindigkeit der FPGA-basierten Fehleremulation zu schließen. Ich präsentiere Optimierungen, die den Kommunikationsflaschenhals zwischen Soft- und Hardwarekomponenten der Fehleremulationsumgebung beseitigen, die Laufzeit für Fehlerexperimente reduzieren und Fehlerexperimente abbrechen, wenn diese Fehlereffekte zeigen, die bereits in vorhergehenden Experimenten beobachtet wurden.

Um die Verbesserung gegenüber dem Stand der Technik zu diskutieren, analysierte ich die Schaltungsdesigns zweier Sicherheitskontroller, für welche ich einige hundert Millionen verschiedener Fehlerattacken emulierte. Die Ergebnisse meiner Experimente zeigen, dass die präsentierten Fehleremulationstechniken, im Gegensatz zum Stand der Technik, die Modellierung be-

liebiger Fehlerattacken in kombinatorischer und sequentieller Logik industrieller Schaltungsdesigns unter Verwendung repräsentativer Tests und im Vergleich zu einer fehlerfreien Schaltungsemulation ohne Geschwindigkeitseinbußen ermöglichen. Für die Fehlerinjektion in kombinatorischer Logik konnte ich, verglichen mit bisherigen Ansätzen, den benötigten Hardwareaufwand um 45% reduzieren und die Laufzeit um den Faktor sechs reduzieren. Durch die Reduktion des Hardwareaufwands können größere Schaltungen verifiziert werden. Mit Hilfe der präsentierten Fehleremulationstechniken lassen sich beliebige Fehlerattacken im Rahmen der Sicherheitsverifikation während der Schaltungsentwicklung analysieren, wobei zusätzlich mehr Fehlerattacken in gegebener und üblicherweise limitierter Zeit modelliert werden können. Diese Arbeit trägt daher dazu bei, die Sicherheit integrierter Schaltungen zu verbessern, da mehr Sicherheitsprobleme bereits während der Schaltungsentwicklung, also bevor die gefertigte Schaltung Post-Silizium zertifiziert wird, ausgeschlossen werden können.

Acknowledgements

I thank my adviser Prof. Dr.-Ing. Georg Sigl for his supervision, support and patience over the past years. I would also like to thank my examiners Prof. Dr.-Ing. Ulf Schlichtmann and Prof. Dr.-Ing. Dirk Rabe for their support.

I am sincerely grateful to Prof. Dr.-Ing. Dirk Rabe, who initiated the HaVerI project in which I started my scientific work and who motivated me to pursue a doctorate.

I am particularly grateful to Prof. Dr.-Ing. Georg Sigl and Prof. Dr. Claudia Eckert for giving me the opportunity to continue my scientific work at Fraunhofer AISEC.

I would also like to thank my colleagues from Infineon Technologies AG, Fraunhofer AISEC, Technische Universität München and Hochschule Emden/Leer for collaboration on scientific and industrial projects as well as scientific exchange. My special thanks goes to Dietmar Heinz, Prof. Dr.-Ing. Gerd von Cölln and Dr.-Ing. Johann Heyszl for giving me opportunities, to Jürgen Nolles for his support, to Robert Hesselbarth for inspiring discussions in the final phase of writing my dissertation and to Robert Specht for proof-reading my dissertation.

SPONSORED BY THE

Federal Ministry of Education and Research

Parts of this work originate from the project *HaVerI (FKZ 17N1509)*, which was funded by the German Federal Ministry of Education and Research.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Connected electronic devices are essential in our daily routines. To name just a few examples, we use smart phones, laptops and personal computers for checking daily news, writing emails or messages and managing the appointments of the upcoming day. Our favorite TV series are provided by streaming video on demand services, for which also TVs and Blue-Ray players are connected. Nowadays, we expect getting access to the Internet with our smart phones in public networks from, e.g., cafés and public transportation. The number of sold smart phones in 2016 is about 1,5 billion [Gar17b], which clearly indicates the strong impact of these devices on our daily live.

Hot topics in the information technology related research domains are today Internet of Things, 'Industrie 4.0', autonomous cars, smart homes and smart grid. In these emerging areas many different devices and machines are connected to exchange information, building a system that provides a service. In contrast to the listed devices above used by human beings to communicate or to enjoy media and multimedia, the incorporated devices work autonomously, i.e. without interaction with human beings. This is the reason for a heavy increase of connected devices in the last decade, which will still drive the market in the next decade. In 2015, the analysts of Gartner, Inc. [Gar15] predicted that 6.4 billion connected devices would be in use worldwide by end of 2016. These numbers are confirmed now and 8.4 billion connected devices are predicted to be in use by end of 2017 [Gar17a]. This corresponds to an increase of about 30 percent each year. Correspondingly 20.4 billion connected devices are expected in 2020 [Gar17a]. This tremendous amount of devices introduces huge potentials for adversaries to perform on-line attacks, exploited to break into systems and networks in order to violate privacy, confidentiality and integrity. In case of online attacks, the victim may realize that he is being attacked, enabling him to possibly counteract this threat before an adversary may cause any damage. Even more powerful are therefore off-line attacks, where attacks are repeated on isolated devices until its success without anyone noticing it. Successful attacks can then be replayed as on-line attacks, which increases the probability to compromise a system before respective attacks are detected by the victim. For example, secret information such as cryptographic keys and passwords can be extracted off-line, which then can be exploited to compromise systems by means of accessing their secured interfaces or breaking secured communication channels.

Systems are not appropriately secure by simply using encryption to protect sensible data. Even if the utilized encryption algorithm is theoretically secure, the hardware or software implementation of the cryptographic algorithm itself may introduce leakage of information about the processed secrets due to for example varying execution time, power consumption or electromagnetic emission of a circuit (also known as side channels). Basically, performing repeated passive measurements of these side channels a processed cryptographic key could be extracted. There is another class of attacks that exploits deliberately injecting faults into a system, where the fault may cause a malfunction or service failure, i.e. the circuit behavior deviates from the circuit's specification. This way, very critical behavior can be caused, e.g., disturbing the system such that its availability is violated, which would be critical in numerous safety applications. Especially critical for security applications would be to exploit faults to circumvent specific checks on, e.g., passwords to cause buffer overflows or to disrupt a cryptographic algorithm such that it outputs the processed key instead of the ciphertext.

Consequently, various attacks possibly mounted on systems and devices need to be considered when designing security circuits and security systems. Therefore, to protect sensitive data in terms of confidentiality as well as integrity and to enable secured authentication, it is not sufficient to rely solely on deploying cryptographic primitives. In addition, appropriate countermeasures need to be deployed that counteract passive side-channel attacks and active fault attacks. Basically, fault countermeasures introduce redundancy in hardware or software that enables to detect corrupted behavior caused by fault attacks and then actively prevents that an attack can be exploited. This is the purpose of security circuits, which include relatively complex processor-based circuits such as so-called secure elements, trusted platform modules (TPM), hardware security modules (HSM) and smart cards. These allow secured key storage, allow to perform cryptographic operations in untrusted environments and provide secured code execution, used to protect sensitive data and its integrity and to realize secured authentication. For this purpose, security circuits deploy cryptographic peripherals such as hardware accelerators for cryptographic algorithms and random number generators. These

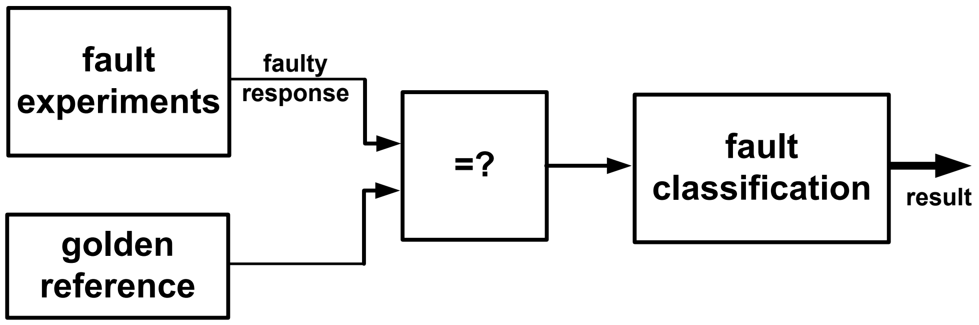


Figure 1.1: Concept of fault injection techniques.

are implemented and integrated into circuits in a secure way, for which it is necessary to deploy additional countermeasures dedicated to specific passive and active attacks. To create a secured system, any incorporated security-critical device that is accessible to adversaries need to be designed in a secure way.

In this thesis the focus is on security verification, particularly on verification of the effectiveness of countermeasures that are dedicated to active fault attacks. In order to check the effectiveness of fault countermeasures, security verification is required to be performed in addition to common functional verification. Since fault countermeasures are only supposed to take any action in presence of faults, various fault attacks need to be mimicked during security verification by means of performing a huge number of fault experiments, during which faults with varying spatial and temporal properties are injected into the circuit under verification.

Figure 1.1 depicts the basic concept of fault injection techniques, which I am going to apply to security designs in this thesis. During *fault experiments*, the circuit to be verified needs to be sensitized by appropriate tests, i.e. input pattern or, in case of processor-based architectures, test software. The purpose is to represent relevant situations such as performing encryption and decryption using a cryptographic algorithm implemented in hardware or software, while attacks are being mimicked. In case of processor designs, usually relative extensive tests lasting thousands of clock cycles are applied since deployed software-based countermeasures need to be considered as well. Due to the fault injection, the circuit may output a *faulty response*, which is compared to the fault-free response generated by the *golden reference*. If a critical malfunction is observed that is not detected by fault countermeasures, faults are classified as critical. These are the cases of interest, which suit as starting point for further investigation with the goal to track the issue.

Important to note is that fault injection can be applied to any level of abstraction of the circuit under verification. This includes pre-silicon techniques that are already applicable during circuit design and post-silicon physical fault injection into the physical device. As for design bugs, it is important that vulnerabilities and security leaks are identified as early as possible during circuit design in order to save design costs and reduce time-to-market. Thus, pre-silicon fault injection tools are required that are applicable during circuit design.

1.1 Problem Statement

Unfortunately, there is only a limited time frame for security verification during circuit design. Furthermore, due to the complexity of the verified circuits and applied tests and also due to the complexity of the fault injection space, it is impossible to consider all possible configurations for fault injection. In fact, only a very small subset of the total fault injection space can be considered within the given time frame for security verification. This motivates high speed tools for fault injection that are also required to provide the configurability required to mimic arbitrary fault attacks. Moreover, these tools are required to be applicable to relatively complex circuits and are required to handle extensive tests.

Pre-silicon fault injection tools based on software implementations including statistical, analytical, probabilistic and symbolic techniques as well as simulation-based, emulation-based and hardware-based (e.g. development boards) implementations are reported in literature (detailed in Section 3.6). All fault injection tools have application specific merits and strength. From these I choose the FPGA-based fault emulation technique, particularly the circuit instrumentation technique, which alters the design during security verification to add fault injection capability. The instrumented design is then synthesized into an FPGA. Additional software components are added to configure and control fault emulation. The biggest advantage over all other approaches is that FPGA-based fault emulation is the fastest method to perform fault propagation in a circuit after fault injection. Contrarily to post-silicon physical fault injection techniques, for which the circuit needs to be already manufactured, this technique can be applied early during circuit design. The performance of fault emulation is linear with the duration of the test that sensitizes the circuit under verification. Therefore, fault emulation provides high speed even for extensive tests, whereas software-based and simulation-based approaches would struggle.

Of course, fault emulation comes with some disadvantages. Compared to software-based and simulation-based approaches, fault emulation is considered to provide less configurability and less observability. These two disadvantages have a considerable impact on applicability of fault emulation during security verification. High configurability is a requirement to mimic arbitrary fault attacks. So far, higher configurability comes with considerably reduced performance because of a communication bottleneck between controlling software components and hardware components. Observability is important to reliably identify vulnerabilities of security designs. For this purpose, not only the functional behavior of the circuit under verification needs to be checked, but also the effectiveness of deployed fault countermeasures, which is especially challenging when processor designs with limited top level interfaces are verified. Another drawback of FPGA-based emulation is that hardware resources available on FPGAs are limited. For mimicking arbitrary fault attacks fault injection capability needs to be added to every incorporated cell, for which additional logic is required. This limits the size of the circuit under verification (CUV), and therefore, applicability especially when considering fault injection in combinational cells.

1.2 My Contribution

The goal of this thesis is to eliminate the discussed disadvantages of FPGA-based fault emulation. I maximize the configurability for mimicking arbitrary fault attacks while the performance is maximized at the same time, which allows to benefit as much as possible from limited verification times. I decrease the hardware requirements for fault injection in combinational logic, and thus, increase the applicability in general. I increase observability for security controllers by introducing concepts for reliable response observation and fault classification, which at the same time save hardware and runtime overhead. In particular, I contribute the following improvements to the state-of-the-art, some of which have been partly or fully published in [NR11, NHN$^+$14, NHRS15, NHS15, NHHS16]:

- I present an FPGA-based fault emulation environment that allows to configure arbitrary permanent and transient multiple faults at runtime without requiring to re-synthesize the fault emulator. This is a requirement for mimicking arbitrary fault attacks.

- I present a feature for fault emulation that allows to configure multiple fault injection times, where for every clock cycle of a test an arbitrary multiple fault can be injected without limitations. This feature completes fault configurability in sequential logic of FPGA-based fault emulation, for which I also propose a very hardware-efficient implementation (not published yet).

- I close the gap between speed and configurability of fault emulation environments by introducing performance optimization measures aiming on fighting the communication bottleneck between software and hardware components. These allow to support the high configurability requirements without performance loss compared to fault-free test runs.
- I propose additional performance optimization measures that aim on reducing the time required for fault experiments and skipping equivalent fault experiments entirely (not published yet).
- I present a generic concept for reliable response observation and fault classification, which is supported by software-based self-tests in case that the circuit under verification is a processor-based architecture. The proposed concept reduces false positives in emulation results, allows to monitor the state of the circuit under verification and supports defining test-related events used to control fault emulation. Thus, observability, controllability and applicability is improved.
- I introduce a new method enhancing fault injection in combinational logic by extending FPGA-based fault emulation by a software-based pre-processing that maps faults from combinational logic to equivalent faults in sequential logic. This method benefits from the high performance that FPGA-based fault emulation provides while limitations of conventional approaches with respect to applicability and performance are eliminated. Further performance optimizations based on skipping equivalent fault experiments are proposed for this method.
- I propose a meta fault configuration model that allows to describe all configuration possibilities required to mimic arbitrary fault attacks independently of the targeted circuit under verification, specific levels of abstraction and the technique used for fault injection. I use it to derive fault models for the considered level of abstraction to provide formal descriptions for all presented techniques and implementations.

1.3 Thesis Structure

Next, in Chapter 2, I present the background to this thesis by means of reviewing the existing work related to physical fault effects, physical attacks and physical fault injection techniques.

In Chapter 3, I briefly outline security requirements and fault countermeasures and introduce fault injection concepts. Then I discuss along with the related work how physical faults are modeled using fault models and I review testability of faults and propagation of transient faults. My intention is to impart the basics of fault modeling, which is used to abstract complex physical behavior, and to impart that fault behavior depends on the sensitizing test. Finally, I review appropriate fault injection tools, where the focus lies on discussing the respective advantages and disadvantages in order to argue my decision to choose FPGA-based fault emulation for modeling fault attacks during security verification.

In Chapter 4 my goal is to create a comprehensive understanding of all possibilities for configuring fault properties during fault injection. Since fault modeling can be applied to any level of abstraction of a circuit under verification, I decided to formulate a fault configuration model at a meta level. It is based on a fault configuration space that covers all configuration possibilities independently of the targeted circuit under verification and its considered level of abstraction. It is independent of specific fault injection tools and techniques as well. This way, the proposed model constitutes a superset of specific fault models known from literature. This formal description is used to derive a fault configuration model at gate level later in Chapter 6, based on which formal descriptions of following concepts and implementations are provided. Moreover, I discuss in the context of the meta fault configuration model terms that are commonly used to describe properties of faults, fault models and fault experiments. I focus on terms that allow to describe specific practice-oriented subsets of the fault configuration space. Finally, equations are derived that allow to determine the complexity of the formulated fault injection space for practice-oriented subsets.

I decided to review the state-of-the-art of FPGA-based fault emulation in an own dedicated chapter, presented in Chapter 5. There I focus on how fault emulation realizes the concepts of fault injection and detail respective methodologies for fault injection, fault generation, test generation and response observation. I also discuss possibilities to implement components of fault emulation environments either in software or hardware, where I focus on respective advantages and disadvantages, based on which I make my design decisions.

Then, in Chapter 6, I present my hardware implementation for FPGA-based fault emulation that I have chosen to maximize configurability. My implementation provides configurability at runtime for arbitrary permanent and transient faults including single and multiple faults, at runtime configurable fault duration and at runtime configurable fault model types. After this, the hardware implementation is embedded into a fault emulation environment, which adds software components to complete the concepts and methodologies of FPGA-based fault emulation discussed earlier when reviewing the state-of-the-art. To provide formal descriptions for the presented techniques and implementations, I apply the meta fault configuration model to gate level by means of refining its properties accordingly. Finally, I present advanced concepts for response observation and fault classification that suits security designs. These concepts are supported by software-based self-tests in case that the circuit under verification is a processor-based design.

Chapter 7 is dedicated to fault injection in combinational logic. There, I detail limitations w.r.t. applicability and performance that are encountered when using FPGA-based fault emulation for fault injection in combinational logic. In order to eliminate these limitations, I present a concept as well as an implementation that extends fault emulation by a software-based preprocessor for fault injection and fault propagation in combinational logic. Basically, I separate combinational and sequential fault propagation, where combinational fault propagation is performed in software in order to determine an equivalent fault in sequential logic. Then, the fault emulator presented earlier is used for further sequential fault propagation by means of injecting the determined equivalent fault into sequential logic. The concept is supported by reviewing the literature related to state equivalence and fault equivalence, based on which I derive the relation of equivalent transient faults.

In Chapter 8, I close the gap between speed and configurability of fault emulation environments by means of introducing performance optimization measures. These allow to reach optimal performance while supporting the ability to configure arbitrary faults at runtime. The presented performance optimization measures aim on fighting the communication bottleneck between software and hardware components of the fault emulation environment. Moreover concepts for shortening fault experiments and skipping equivalent fault experiments are proposed. Then, I propose a feature for enabling multiple fault injection times. This feature enables to configure any fault defined in the fault configuration space of the meta fault configuration model.

In Chapter 9, I provide experimental results for fault injection campaigns. First, I concentrate on fault injection in sequential logic to analyze the proposed performance optimizations for fighting the communication bottleneck of fault emulation environments. After this, I present results for fault injection in combinational and sequential logic. There, I focus on fault propagation results to discuss the efficiency and effectiveness of the performance optimizations proposed for the software-based pre-processing for enhancing fault injection in combinational logic. Then, I discuss respective performance results and provide a comparison to the state-of-the-art. I am able to show that the proposed techniques provide a highly configurable and fast tool to perform fault injection in both combinational and sequential logic, which suits the purpose of security verification. Moreover, the presented techniques remove the limitations of state-of-the-art approaches w.r.t. speed, configurability, observability and applicability.

Finally, Chapter 10, summarizes and concludes this thesis.

Chapter 2

Attacks on Integrated Circuits

In the last decades, many efforts have been made on studying the reasons for fault occurrences, their impact on circuit behavior and modeling fault behavior. A lot of studies were motivated by manufacturing and quality testing whose goal is to find efficient tests, mainly used to detect permanent defects in integrated circuits. Advances in circuit design and semiconductor processes result in shrinking technology nodes and increasing operating frequencies. This leads to more frequent occurrence of transient faults and increases their impact on circuit behavior. As a consequence, reliability engineering emerged, who pushed research in the direction of transient faults with the goal of hardening circuits against random fault occurrences during in field operation. Nowadays, reliability engineering is considered to be an important design parameter in addition to the conventional power-area-performance trade-off, not only for application in very harsh environments like outer space and safety-critical application domains (e.g. automotive and avionics), but also for consumer products.

This work is dedicated to security engineering, which is another important application domain that is concerned with faults. In a security context, not only random fault occurrences have to be considered and countered by e.g. error detection and correction codes. Additionally, deliberately injected faults aiming on compromising security applications during in field operation, so called fault attacks, are one of the major concerns for concept, design and verification engineers of security circuits. Therefore, fault countermeasures dedicated to specific fault attacks are deployed, which need to be verified during circuit design.

Next, I outline the taxonomy of attacks on integrated circuits in Section 2.1 to define fault attacks, which is the class of attacks to which this thesis is dedicated. In order to draw a comprehensive picture, I briefly sketch the historical context on faults in integrated circuits in Section 2.2.1. In this thesis, I make use of definitions and techniques that are mostly originated from application domains that are not dedicated to the security context, e.g. manufacturing testing and reliability engineering. I therefore discuss in the remainder of Section 2.2 fault effects in integrated circuits along with the related work from a general point of view and introduce related definitions. Various physical fault injection techniques that have been successfully exploited for fault attacks to break cryptographic algorithms as well as their respective physical effects are reviewed in Section 2.3.

In Chapter 3, I bridge the gap to fault countermeasures and fault injection concepts, including common fault models, testing concepts and fault classification. Additionally, I review pre-silicon methods capable of fault modeling.

2.1 Taxonomy of Attacks

Attackers try to compromise security circuits and cryptographic algorithms by means of conducting attacks in order to reveal secrets. In general, attacks are divided into active and passive attacks, as defined in [Shi00] and discussed by Skorobogatov et al. [Sko05] and Mangard et al. [MOP07].

Passive Attacks Passive attacks observe physical properties (side channels) of a circuit such as operating temperature, power consumption or execution time while the circuit is operated in its specification. In literature, these attacks are also referred to as side-channel attacks.

Active Attacks Active attacks manipulate a circuit or disrupt its function in order to prompt an abnormal circuit behavior that does not comply with the circuit's specification (service failure). For this, the circuit itself can be altered, an extra amount of energy can be induced or environmental and operating conditions that do not comply with the circuit's specification can be exploited. Injecting faults deliberately, also known as fault attack in literature, belongs to the class of active attacks.

Skorobogatov et al. [Sko05] and Mangard et al. [MOP07] subdivide passive and active attacks further into invasive, semi-invasive and non-invasive attacks, which indicates whether and how much a circuit is altered for conducting an attack. Although I focus on active attacks (fault attacks) in this thesis, the following description will also consult definitions for passive (side-channel) attacks.

Invasive Attacks Invasive attacks depackage a circuit to expose the die by grinding or by using laser cutters or acid in order to apply further modifications. For example, focused ion beam (FIB) and probing stations are often used to contact internal signals. According to Mangard et al., using probing to observe internal signals is a passive attack, whereas changing internal signals with this setup is an active attack.

Semi-invasive Attacks Semi-invasive attacks only depackage a circuit to expose the die but do not modify the die itself. This is required to conduct active attacks using e.g. a laser or light as well as optical passive side-channel attacks. It also improves precision of active attacks that use radiation sources (alpha particles, electromagnetic interference, etc.) and passive electromagnetic side-channel attacks, since the radiation source or probe can be put closer to the surface.

Non-invasive Attacks Non-invasive attacks do not alter a circuit and are conducted by exploiting directly accessible interfaces, environmental and operating conditions, radiation sources or side channels. Active non-invasive attacks can be induced by clock glitches (clock signal variation), power supply glitches (supply voltage variation), temperature variation of the environment and radiation sources such as alpha-particle and electromagnetic interference. Passive non-invasive attacks include timing, power, temperature, photonic and electromagnetic side-channel attacks.

2.2 Basics of Faults in Integrated Circuits

In this section I start with the historical background on faults in integrated circuits (IC). Then I give definitions related to faults, errors, failures and testing for faults, after which I bridge the gap to physical fault attacks.

2.2.1 A Historical Perspective

Faults in digital integrated circuits may occur because of different phenomena. For instance, manufacturing defects resulting from defective fabrication and process variation is one cause of fault occurrences. Rejecting defective devices to prevent that these are shipped to customers opened the oldest research area concerning faults, manufacturing testing, which includes disciplines such as test generation and fault simulation. In these research fields, relevant faults are usually assumed to be permanently present in circuits.

However, back in the 70's faults were discovered that exhibit transient characteristics. The relevant historical facts outlined in the following paragraph are inline with the book Soft Errors in Modern Electronic Systems [Nic10].

In 1975, Binder et al. [BSH75] reported on anomalies in space applications that triggered flip-flops. These effects were observed about one time in four years and were not caused by manufacturing defects. These could not be explained by known effects such as solar wind either. Instead, Binder et al. concluded that cosmic ray particles with high atomic numbers and high energy produce dense ionization tracks of electron-hole pairs, which charges base-emitter capacitance of transistors. In 1978, May and Woods [May78] reported that soft-errors also occur at sea-level. These were caused by alpha-particle hits emitted from radioactive decay of uranium and thorium used in package materials. May and Wood also introduced the term soft-error as random, nonrecurring single-bit error. In the same year, Ziegler suggested that cosmic radiation possibly may also cause soft-errors. In the 1990s, when package materials with low emission rates were already used, and therefore, could not be the reason for soft-errors, it was found that cosmic neutrons pose the main source for soft-errors [Zie96].

Nowadays, it is known that many sources are capable of inducing enough energy to cause a transient abnormal circuit behavior. These include, radiation sources, high-energetic particle hits and electromagnetic interaction, as will be detailed along with the literature in following sections. Because of shrinking technology nodes and growing device count per chip, particle hits occur more frequently [Bau05, MM07]. First studies focused on fault effects on sequential cells. However, reduction in operating voltage and increasing operating frequencies increase the probability that particle hits in combinational logic result in errors [Bau05, MM07]. As a consequence, the impact of faults in combinational logic could not be neglected anymore. Moreover, because of further advances in circuit design and manufacturing technologies, particle hits causing multiple faults are more likely to occur and it became important to study also these effects [MZM10].

2.2.2 Causality Relationship of Fault, Error and Failure

A fault has the potential to manifest as an error in an internal state of a system, which in turn may cause a service failure observable as corrupted external state, as discussed e.g. by Clark et al. [CP95] and later refined by Avizienis et al. [ALR01, ALRL04]. This causality relationship is often referred to as fault-error-failure chain [ALR01], which is depicted in Figure 2.1.

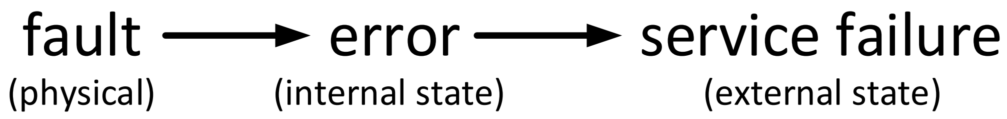


Figure 2.1: Fault-error-failure chain, illustrating the causality relationship of fault, error and failure.

The terms fault, error and failure are defined referring to the literature as follows:

Fault A fault is a physical defect, imperfection, flaw or abnormal event. Faults can occur in hardware or software at any level of abstraction and during the entire life cycle of a system [ALRL04] ranging from specification and design over manufacturing to in field operation.

In this thesis, I focus on modeling hardware faults and I use the term fault for abnormal circuit behavior caused by a physical effect or fault injection. The exhibited behavior of a hardware fault depends on its cause, the affected area and its temporal properties timing of occurrence, timing of dormancy relative to input or software execution, from which follows the fault duration.

Error An error is the deviation of correct and incorrect internal system states. Faults that manifest in at least one memory element result in an erroneous internal state, and hence, implicate an error [ALRL04].

In the literature, the two terms hard-error and soft-error commonly refer to permanent and transient errors, respectively. The term soft-error was originated by May et al. [May78] to described fault effects in dynamic memory cells.

Failure The term failure is often used as abbreviation of service failure and refers to the deviation of delivered incorrect service from correct service [ALRL04]. A system's service corresponds to the system's external state, i.e. the system's output, which is generated by a sequence of the system's internal states. Therefore, a service failure is the implication of at least one erroneous internal system state [ALRL04].

Note that, in a security context, a service failure occurs when assets or secrets such as sensible data or cryptographic keys are exposed to adversaries. This happens, for example, when fault countermeasures supposed to detect specific attacks in order to protect the defined assets and secrets actually fail. Fault attacks and countermeasures are further detailed in Sections 2.3 and 3.1.

Soft-Error-Rate (SER) In the domain of reliability engineering the Soft-Error-Rate (SER) is an important measurement for quantifying the vulnerability to random fault occurrence. It expresses the frequency of random soft error occurrence caused by e.g. particle hits [PHRB11], given in failure in time (FIT), which is the number of failures that can be expected in one billion device hours of operation.

2.2.3 Persistence of Faults

Based on the duration faults exhibit, Clark et al. [CP95] divide faults into permanent, intermittent and transient faults. Avizienis et al. [ALR01, ALRL04] suggest to use the term persistence as hypernym for these properties.

Permanent Faults Permanent faults are irreversible device defects, which are permanently effective and are caused by e.g. damage (mechanical, overvoltage, etc.), fatigue or incorrect manufacture [CP95]. Permanent faults can be further subdivided into static and dynamic faults. Static faults include electrical shorts (stuck) to power supply. The fault behavior of static faults is independent of the operating clock frequency. Dynamic faults include delay faults, which are only effective with frequencies above a specific value.

Intermittent Faults Intermittent faults in integrated circuits are irreversible device defects which tend to oscillate between periods of erroneous activity and dormancy [CP95], i.e. these are temporarily present and a non-deterministic appearance is typical. Intermittent faults can be caused by e.g. erroneous design [CP95], irregular physical structure of components, critical circuit timing, stray capacitances, aging, fatigue, noise, loose connections [KP74], electromigration [KE15] and tunneling effects [Con03]. Faults often exhibit an intermittent characteristic before they turn into a permanent fault, caused by, e.g., a permanent oxide breakdown [Con03].

Transient Faults Transient faults are caused by reversible single or multiple event effects (SEE, MEE). These occur infrequently and temporarily when either an extra amount of energy is added into a circuit or when its environmental or operating condition are varied in a range that does not comply with the circuit's specification. Transient faults cause an abnormal electrical behavior for a relatively short period of time (e.g. induced voltage pulse) and usually do not damage a circuit. Note, in rare cases when the energy is high enough, it may result in a latchup or burnout. Both latchups and burnouts may cause permanent damage [Bau05], i.e. single event effects may also result in permanent faults or permanent errors.

Sources for transient faults include radiation such as neutrons [Zie96, KHP04], alpha particles [May78, KHP04] and electromagnetic interaction [QS02, MLB$^+$14]. Furthermore, environmental and operating conditions capable to influence electronic properties temporarily, e.g. variation of temperature [Sko09, KJP14], operating frequency [FT09] and operating voltage [ABH$^+$02] may cause transient faults. The caused abnormal faulty behavior depends on the affected area (spatial property) and, in contrast to permanent faults, especially on the induced energy, the circuit's current state as well as the timing and duration of fault occurrence relative to the circuit's input or software execution (temporal properties). A very comprehensive work on transient fault effects is presented by Karnik et al. [KHP04].

Different terms are used in literature to describe transient fault effects based on single and multiple event effects, to distinguish between single and multiple fault occurrences and to distinguish whether combinational or sequential cells are affected. Unfortunately, theses terms are sometimes used inconsistently and interchangeable [PHRB11] in an ambiguous way. Moreover, these terms are used to name the physical fault effects, but are also used as synonym for related fault models. This makes it difficult to clearly distinguish between the actual physical effect and the describing fault model.

The following terms are used hereinafter to describe modeled transient fault effects.

- *Single Event Upset (SEU)* where a single transient fault is present in a sequential cell (single upset).
- *Multiple Event Upset (MEU)* where multiple transient faults are present in sequential cells (multiple upsets).
- *Single Event Transient (SET)* where a single transient fault is present in a combinational cell (single transient).
- *Multiple Event Transient (MET)* where multiple transient faults are present in combinational cells (multiple transients).

In the literature, the following synonyms are sometimes used. Multiple Cell Upset (MCU) or Multiple Bit Upset (MBU) are typically used as synonym for MEU, where MBU is then usually used to describe the case that multiple faults affect the same word. Single Bit Upset (SBU) is sometimes used as synonym for SEU. SEU is often used as synonym for soft-errors in general and sometimes referred to as the superset of SBU, MCU, and MBU.

To distinguish between single and multiple event effects, sometimes terms were introduced that emphasize on that matter. For example, Kiddie et al. [KRL15] recently denoted fault effects stemming from single and multiple event effects as Single Event Single Transient (SEST) and Single Event Multiple Transient (SEMT), respectively. In contrast, Miskov-Zivanov et al. [MZM10] use the terms Single Event Multiple Transient Fault (SE-MTF) and Multiple Event Multiple Transient Fault (ME-MTF). In order to keep the nomenclature compact, I relinquish to distinguish between single and multiple event effects for the reminder of this work.

2.3 Fault Attacks

The objective of this work is modeling fault attacks using emulation techniques. Next, an introduction to fault analysis at algorithm level and methods for physical fault injection are given. I focus on techniques that have been successfully mounted at circuit level to break cryptographic algorithms.

2.3.1 Fault Analysis at Algorithm Level

Incorrect ciphertexts or signatures produced by a disturbed cryptographic algorithm in response to a fault attack may already pose a huge security weakness, which can be analyzed at algorithm level. By analyzing the deviation of correct and incorrect ciphertexts, for which Biham et al. [BS97] introduced the term differential fault analysis (DFA), a cryptographic key can be extracted. Boneh et al. [BDL97] were the first to present a theoretical model for exploiting fault attacks and were able to break a public-key cryptographic algorithm, namely Rivest Shamir Adleman (RSA). Similar attacks were deployed to private-key cryptographic algorithms. Biham et al. [BS97] deployed it to the Data Encryption Algorithm (DES), Biehl et al. [BMM00] to elliptic curves and Girau, Dusart et al., Blömer et al. and Piret et al. [Gir03, DLV03, BS03, PQ03] deployed it to the Advanced Encryption Standard (AES).

The above listed publications provide theoretical aspects of differential fault analysis, where the presence of faults is only assumed during the execution of cryptographic algorithms. To actually mount a successful fault attack onto a physical device, this fault needs to be injected into the physical device and information about the used secret is obtained from a corrupted output. In-depth knowledge about the theoretical background on differential fault analysis helps to select the timing and location for an attack mounted on the physical device, i.e. at circuit level. Subsequently, methods for physical fault injection and related practical setups are discussed.

2.3.2 Methods for Physical Fault Injection

Note, methods for physical fault injection discussed in the remainder of this section are also used for post-silicon verification of security circuits.

Physical fault attacks are active attacks mounted at circuit level to prompt a fault in the circuit. The underlying physical effects of known methods for physical fault injection differ. Relatively prominent surveys on fault attacks providing a comprehensive overview and insight on techniques and physical effects are presented by Bar-El et al. [BCN$^+$04] and Giraud et al. [GT04].

Injecting faults deliberately into a circuit may change circuit behavior in such a way that access to sensible data is granted or secrets are leaked. In the worst case, the attacked device outputs a secret key instead of the ciphertext, which was reported for example by Trichina et al. [TK10]. In general, the success of fault attacks increases by focusing on very small, local and specific regions of a circuit (e.g. only affecting transistors of a specific set of cells) and by controlling the timing precisely. Usually, it is desired to not damage the circuit with an fault attack, since fault attacks have to be repeated with varying parameters to succeed. Expertise in the circuit's layout and functionality helps to successfully conduct fault attacks.

Various methods for physically mounted fault attacks are reported in the literature. These differ mainly in precision w.r.t. temporal (timing) and spatial granularity (locality), acquisition cost, availability, replication ability and required expertise for their application. The most relevant methods for fault attacks are optical attacks, such as laser beams and intense light. Recently, electromagnetic (EM) glitch attacks gained increasing attention. Supply voltage, clock signal and temperature variations are also very prominent sources for fault attacks, since these were the first techniques reported in literature. X-rays and alpha-particle attacks are also reported in literature but these attacks are uncommon, and I concentrate subsequently on the other, more common fault attack methods. Fault effects of high energetic particles is a well studied topic in nuclear physics and reliability engineering, and therefore I refer to the related literature, e.g. [KHP04] and [Bau05].

In the following, I briefly review the most relevant physical fault attacks that have been successfully mounted on physical devices. I elaborate on the temporal and spatial precision of particular attacks, which I take up again in Section 3.3 while reviewing appropriate fault models.

Optical Attacks Optical attacks pose the most common method for physical fault injection, which include attacks utilizing intense light (e.g. flashgun) and laser-beams. Since the package of the chip has to be removed, so that the chip surface is exposed, optical attacks belong to the class of semi-invasive attacks.

Integrated circuits are sensitive to light due to photo-electric effects, which is exploited by optical attacks. Basically, based on photo-electric effects the generation of electron-hole pairs in the semiconductor is caused [LNF$^+$14]. As discussed in detail by e.g. Wang et al. [WX11], electron hole pairs affected by an electric field result in a transient current pulse. The current pulse loads the gate's capacitance, which in turn generates a voltage pulse. At logical level, the voltage pulse transforms into a signal transition, which propagates through the combinational network. Optical fault attacks can be mounted on both surfaces of a chip, the frontside and the backside. Respective attacks are often referred to as frontside attacks and backside attacks. Castro et al. [CDR$^+$16] recently compared both techniques w.r.t. exploitable errors induced into an AES hardware implementation. Due to the mirror effect of metal layers on the frontside, backside attacks can be (but do not necessarily have to be), more effective because of the freedom to target any desired location [CDR$^+$16].

Skorobogatov et al. [SA02] announced a successful attack using low-cost equipment (flashgun and laser pointer). Schmidt et al. [SH07] mounted another successful low-cost attack using fiber-optic light guides. More sophisticated and very precise but also expensive setups utilize laser beams. Depending on the laser type, wavelength, spot size and duration of the shot, the precision ranges from very short induced transient pulses (magnitude of picosecond) to long transient pulses (magnitude of millisecond), where the laser spot can effect a single gate but also multiple gates with very high multiplicity [VML$^+$14]. Various setups have been reported in literature. For example, Trichina et al. [TK10] were able to break a protected software implementation of a CRT-RSA cryptographic algorithm. For this, they conducted a 'two-fault' fault attack, where two faults were injected with a time offset (also referred to as second-order attack). New multi-beam laser equipment allows to additionally attack multiple locations either simultaneously or with a time offset. This demonstrates the high precision both in timing and locality achieved with laser setups.

Operating and Environmental Condition Variation Attacks Operating condition variation attacks are conducted by exploiting accessible interfaces, i.e. external clock source and power supply, and belong therefore to the class of non-invasive attacks. An investigation in fault attacks based on supply voltage variation was reported by Selmane et al. [SGD08], who reduced the supply voltage during operation. A reduced supply voltage causes affected gates to react more slowly and, thus, increases gate delays. As a result, either timing violations may occur or the data signals may arrive too late at sequential cells and are, hence, not latched, causing transient faults.

More sophisticated setups inject short spikes (rapid transients) into the power network. Basically these also result in timing violations but for a short period of time and therefore provides a better timing precision. These attacks are also referred to as power and clock glitching attacks in the literature. Such an attack was presented by Aumüller et al. [ABH+02], who were able to break an RSA cryptographic algorithm implemented on a smartcard and protected by a software countermeasure.

In case of clock glitching, either clock glitches are inserted into the clock network [FT09] or transient overclocking is utilized [ADN+10]. Similar to power glitching, timing violations may occur, which may transform in upsets in sequential logic. Insides about clock glitching attacks were reported by Anderson [AK96] and practical setups conducted to break an AES cryptographic algorithm implemented in hardware were presented by Fukunaga et al. [FT09] and Agoyan et al. [ADN+10].

Glitching attacks affect power and clock networks globally. It is very likely that gates of the entire circuit or large circuit blocks are affected at once. However, since timing violations cause the faulty behavior, the longest propagation paths are affected the most. This can be exploited to narrow fault effects, and as reported by Maistri et al. [MLB+14], timing violations seem to hit bus transfers first because of their low slack. Agoyan et al. [ADN+10] reported that they were able to precisely create single-bit faults.

Attacks exploiting variation of environmental conditions such as temperature variations (e.g. local heating [Sko09]) also change timing behavior of integrated circuits. The resulting fault effects are very similar to that of supply voltage and clock signal variations.

Electromagnetic Glitch Attacks Electromagnetic glitch attacks are emitted from a probe which is put closely to the surface of a chip. Quisquater et al. [QS02] reported to disrupt the behavior of embedded memories by means of exploiting electromagnetic glitches. Schmitd et al. [SH07] mounted an electromagnetic glitch attack onto a CTR-based RSA cryptographic algorithm implemented in software, and Maistri et al. [MLB+14] presented a comprehensive survey on this matter. There are also successful attacks on hardware implementations reported in the literature. For instance, an attack on an AES hardware implementation was presented by Dehbaoui et al. [DDRT12].

Moro et al. [MDH+13, MDH+14] reported that electromagnetic glitches seem to affect power networks locally. As a result, timing constraint violations are induced, similar to fault effects caused by power and clock glitching. Electromagnetic glitch attacks pose therefore more localized glitch attacks. Due to mechanical limitations of utilized probes, electromagnetic glitches do not reach the precision of laser-induced faults. But as presented by Moro et al. [MDH+13, MDH+14], the precision in both timing and locality is high enough to break an AES cryptographic algorithm implemented in software by means of adding assembly instructions and adding an extra AES encryption round.

2.4 Summary

This section discussed attacks on integrated circuits, where fault attacks, which belong to the class of active semi-invasive attacks, were discussed in detail. This is the class of attacks for which I am going to present fault modeling techniques utilized during security verification, where I mainly focus on emulation techniques. After discussing basics of faults in integrated circuits to create a general understanding of faults and their effects on system behavior, physical fault injection techniques were reviewed. My goal was to show the most relevant fault injection techniques along with successfully mounted attacks reported in literature in order to highlight the relevance of these physical attacks, but also to convey a sense of physical effects caused by fault attacks on integrated circuit. Despite the huge diversity of physical effects, it was shown that basically all fault attacks may cause errors in sequential cells. Errors then in turn may corrupt system behavior, causing service failures, when propagated into external states.

Next, I outline briefly security requirements and countermeasures deployed in integrated circuits to fulfill these requirements in order to protect assets and secrets. Then, after detailing the fundamentals of fault modeling tools including the fault injection concept, common fault models used to abstract physical behavior of faults and fault testing concepts, I review pre-silicon fault modeling tools reported in literature.

Chapter 3

Modeling Fault Attacks during Security Verification of Fault Countermeasures

The effectiveness of countermeasures dedicated to counteract passive or active attacks need to be verified during security verification to ensure that these fulfill the specified security requirements. The focus of this thesis lies on introducing new fault emulation techniques for modeling fault attacks during security verification. I therefore only discuss terms, concepts and methods for security verification w.r.t. to fault attacks. Methods dedicated to security verification w.r.t. passive side-channel attacks are beyond the scope of this work and are not further considered.

Next, in Section 3.1, I briefly introduce definitions for security requirements and the concept of fault countermeasures along with the related nomenclature used in industrial standards like ISO/IEC 15408 (Common Criteria). Then, I give an overview about the concepts of fault injection and introduce related terms in Section 3.2. I detail fault models and fault testing concepts in Sections 3.3 to 3.4, after which I discuss fault propagation and fault classification of transient faults in Section 3.5. Finally, in Section 3.6, I compare common pre-silicon fault modeling tools, based on which FPGA-based fault emulation is chosen as appropriate tool for modeling fault attacks during security verification.

3.1 Security Requirements and Countermeasures

The focus of this thesis lies on techniques used for security verification of fault countermeasures, whereas the design of countermeasures is out of scope of this thesis. However, a brief overview about the purpose and functionality of fault countermeasures is necessary to comprehend the methodologies of security verification. For this, I make use of the nomenclature used in Common Criteria (ISO/IEC 15408), which is the most important industrial standard for certification of security devices in Europe.

Security Requirements As per Common Criteria [Com12] (point 220), a security target (ST) for a target of evaluation (TOE) defines the security problem, assets and threats to those assets. The security target describes the countermeasures in the form of security objectives, which are translated into security requirements in a standardized language. The security target demonstrates that these countermeasures are sufficient to counter the defined threats. Security objectives for the TOE and for the operational environment describe countermeasures that counter the defined threats. Security requirements consists of two groups of requirements (point 383):

- The security functional requirements (SFRs): a translation of the security objectives for the TOE into a standardized language, i.e. a specification of security objectives describing the countermeasures.
- The security assurance requirements (SARs): a description of how assurance is to be gained that the TOE meets the SFRs, i.e. a description of how the TOE is to be evaluated. This includes e.g. testing the TOE and examining various design representations.

In summary, the security target (ST) demonstrates that the security functional requirements (SFRs) meet the security objectives describing the countermeasures for the TOE and that the security objectives counter the threats (point 224 in [Com12]). While SFRs describe the requirements of countermeasures to be evaluated during the Common Criteria evaluation, the SARs define what is done during circuit design (pre-silicon) and at circuit level (post-silicon) to determine correctness of the TOE.

The goal of security verification is to determine correctness of a design in consideration of defined verification conditions during the development cycle of a circuit, i.e. before the design is evaluated for security certification. In case that, for example, circuit parts are identified that are vulnerable to attacks although security objectives describe countermeasures that should protect assets against these threats, design adjustments and security verification are usually repeated until correctness of the design is shown. Therefore, methods used during security verification, including the methodologies presented in this thesis, belong to the Common Criteria SAR.

Fault and Error Countermeasures In general, to prevent service failures, fault and error countermeasures are deployed in all application domains that are concerned with faults. Fault countermeasures counteract either random fault occurrences in e.g. safety-critical circuits or protect security circuits against active fault attacks with the purpose of protecting assets, e.g. preventing the leakage of secrets. For this, redundancy can be deployed in software, hardware or in both hardware and software at all available abstraction levels. The goal is to detect and counteract abnormal operational and environmental conditions, i.e. the cause of faults, and abnormal circuit behavior observable as errors in internal states. The simplest fault countermeasure implemented in hardware is gate duplication, which is routinely used for reliability engineering in order to decrease the Soft-Error-Rate (SER), e.g. proposed in [GJKC06, NJJ06] for combinational logic and in [ZMM$^+$06] for sequential logic.

In a security context digitally designed countermeasures are often deployed, but also analog mechanisms such as sensors and regulators are utilized. Digitally designed countermeasures are used to detect errors in the circuit. Sensors are used to detect the physical cause of faults, i.e. abnormal environmental conditions (e.g. temperature and light sensors) and abnormal operating condition (e.g. clock and voltage regulators). Furthermore, countermeasures implemented in software are deployed to detect abnormal behavior at system level, to support hardware countermeasures and to detect errors in accessible registers and memories.

Often countermeasures implemented in hardware indicate with internal alarm signals the detection of an attack, triggering to actively counteract the attack such that a potential service failure is prevented. In contrast to safety applications, for which it is usually required to recover from a faulty state in order to not violate availability of a service by means of correcting errors, security applications often set the circuit into a secure state such as the reset state to counteract an attack. This way, assets are protected by means of preventing that an attacker gains any advantage out of attacks.

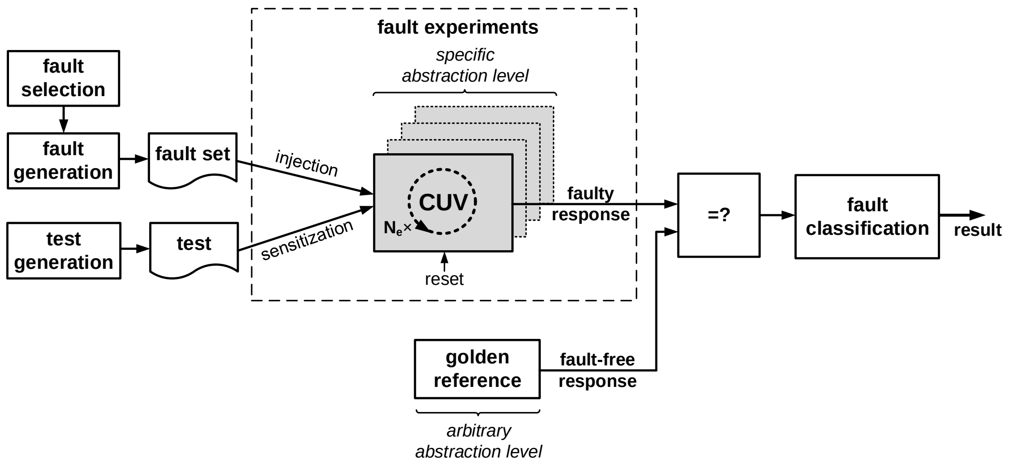


Figure 3.1: Detailed fault injection concept. The dashed area details the *fault experiments* block of Figure 1.1 shown in the Introduction Chapter.

3.2 Fault Injection Concept

Security verification has to consider fault injection because fault countermeasures are only supposed to take any action in the presence of faults. Therefore, security verification applies fault modeling techniques, which mimic an attacker by injecting faults into a circuit representation (e.g. gate level netlist) in order to check whether fault countermeasures work as specified by security objectives.

Next, I give an overview of the fault injection concept, after which I detail the involved disciplines and concepts, namely fault models, fault testing and fault propagation as well as fault classification of transient faults. The following descriptions are not limited to the security context. Instead, these also reference to other application domains that are concerned with faults in order to create a more generalized understanding of the outlined concepts.

Fault Injection Campaign The process of injecting different faults into a circuit and observing its consequences on circuit behavior is referred to as fault injection campaign, which is also referred to as fault injection analysis and fault diagnosis in literature. Fault injection campaigns execute a set of *fault experiments* in which the spatial (locations) and temporal properties (timing) of fault injection are varied so that the fault injection space is covered exhaustively, i.e. all fault configuration possibilities are iterated. Fault injection campaigns are applied in different application domains, e.g. for fault grading of manufacturing tests, verifying fault countermeasures in security and safety-critical application domains and for determining the Soft-Error-Rate (SER) for reliability engineering. The goal of fault injection campaigns is to study the consequences of faults on circuit behavior, to classify the impact of faults, to verify security and safety and to identify most vulnerable circuit parts for reliability engineering.

Figure 3.1 depicts the basic blocks of a fault injection campaign in detail. The dashed area corresponds to the *fault experiments* block shown earlier in Figure 1.1 in the Introduction Chapter. Additional preparation steps are depicted, which include *fault selection* and *fault generation* of to be injected faults and *test generation* of the test processed by the circuit under verification (CUV) during fault experiments. Executing fault experiments is illustrated by the stacked *CUV* block, where the dashed circle indicates that a number ($N_e\times$) of fault experiments are performed during fault injection campaigns. During fault experiments, physical fault effects are abstracted using fault models (detailed in Section 3.3) and the verified circuit is sensitized by a structural or functional test (detailed in Section 3.4). As indicated, the CUV is represented at a specific level of abstraction, where any level of abstraction can be chosen for which also an appropriate fault model has to be considered for fault generation.

Fault Selection Strategies Because of the large fault injection space for multiple faults (detailed in Section 4.4), it is impossible to cover the entire fault space exhaustively. Even if fault models with a high degree of abstraction are used, the fault space for moderate circuits is already too large to cover it exhaustively. In fact, covering all single transient faults in real-world designs requires already millions of fault experiments.

Therefore, fault selection strategies are used to select a relevant subset of the fault injection space that can be handled with fault injection campaigns in a reasonable time. This way, relevant faults that are likely to be caused and that are also relevant considering specified requirements (safety, security, manufacturing testing, etc.) are covered. For this purpose, the physical characteristics of specific faults (defect, particle hit, laser-induced fault, etc.), the interacting physical factors of the circuit under consideration (e.g. circuit technology and layout) and the specified requirements are mapped onto an appropriate fault model.

Computational effort should be spent for mimicking relevant faults, i.e. faults that turn into errors since latched by sequential cells. These faults potentially cause service failures and are therefore especially relevant for checking the effectiveness of fault countermeasures [PHRB11]. Hence, fault injection in sequential cells is commonly performed during security verification, where single event upsets in sequential cells are investigated first before proceeding with relevant multiple event upsets. By concentrating on fault injection in sequential cells, fault injection is applicable to RT level, where it is accelerated with respect to gate level [PHB+14]. More advanced fault selection strategies are considered for multiple fault injection. Appropriate fault selection strategies reported in literature are based on either layout [EAT13, PKE+11] or structural information [PTH+15, PHB+14, VML+14]. Moreover, these strategies can be combined with statistical fault injection [LCMV09] to further reduce the complexity of fault injection campaigns.

Fault injection in combinational logic gained recently increasing attention in the security context. For instance, in [PTH+15] the structure of combinational logic is analyzed to select associated sequential cells for multiple fault injection. However, transient fault injection in combinational logic has been barely tackled exhaustively. This aspect is left out for now and is further detailed in Chapter 7, which is dedicated to fault injection in combinational logic.

Another common fault selection strategy starts at software layer and is used for e.g. security verification of software-implemented fault countermeasures (e.g. [TMS+13]). These strategies limit fault injection to addressable registers and memory of processor designs and do not suit hardware verification.

Fault Generation Based on the output of fault selection strategies, fault configurations are generated defining fault locations, timing, fault duration and the fault model type for fault injection.

Test Generation Tests and test generation are relevant for fault injection campaigns in order to sensitize and propagate faults, which allows to observe their effects. The objectives on tests and test generation are discussed in detail in Section 3.4 within the scope of fault testing concepts.

Fault Classification The behavior of a fault and the consequences for the affected circuit depends on the location where the fault is injected and on the timing related to the workload (input pattern, software, etc.) of the circuit. Analogously to the fault-error-failure chain introduced in Section 2.2, a fault is classified either as a failure when it causes a service failure or as a pass when it does not cause a service failure, as depicted in Figure 3.2. Note that with respect to testing concepts, a detected fault constitutes a failure, whereas an undetected fault constitutes a pass. Fault classification will be

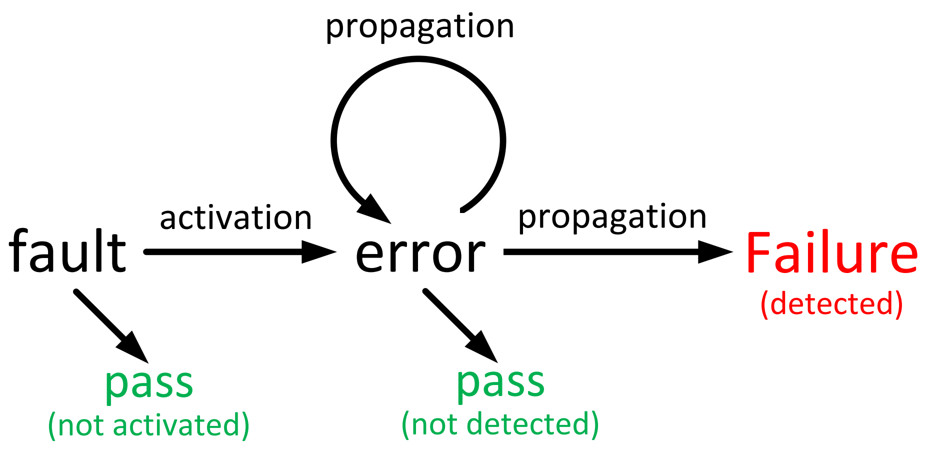


Figure 3.2: Fault classification into failure and passes along with the fault-error-failure chain.

further discussed w.r.t. transient faults in Section 3.5.2 after fault models and testing concepts are detailed.

3.3 Abstracting Faults with Fault Models

In literature, it has been shown that basically all physical fault attacks result in single or multiple faults in sequential logic, even if originated from combinational logic. Temperature, voltage and clock signal variation cause timing violations, which may manifest as upsets in a single or in multiple sequential cells in the fanout cone of the fault location (downstreaming sequential cells) [KJP14, ADN+10, OGST+14]. Laser-induced faults and high-energetic particles cause either transients in combinational logic, which may propagate to downstreaming sequential logic, or single and multiple upsets are caused directly in sequential cells [VML+14, MZM10]. Hence, all physical fault attacks can be mimicked by set, reset or bit-flip fault models. This was stated for laser-induced faults [PHB+14, VML+14, RSDT13], power and clock glitching attacks [ADN+10] and EM glitch attacks [OGST+14]. However, the question is which subset of all multiple faults is appropriate to model these effects. This motivates fault selection strategies, which were discussed in detail in Section 3.2.

Recently, transient faults in combinational logic gained increasing attention, not only for reliability engineering [EAT13], but also for security verification [PTH+15]. On the one hand, there is the need to study e.g.

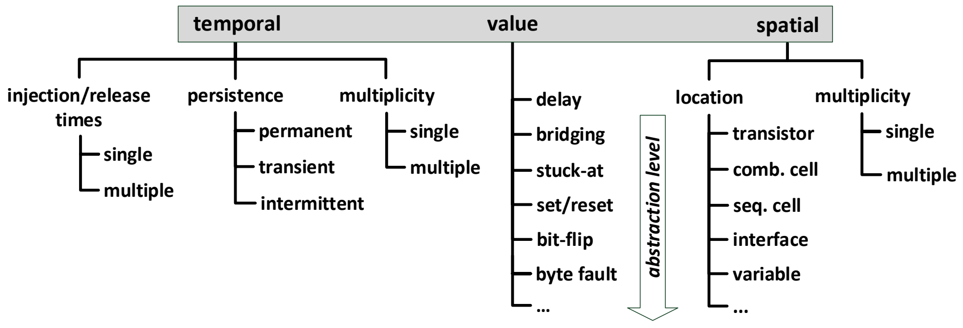


Figure 3.3: Configuration possibilities of fault models, separated into temporal, value and spatial properties.

laser-induced fault effects in order to map these to different abstraction levels [LDCDN+15, PHB+14, PTH+15]. And on the other hand, combinational logic generated and optimized by synthesis and also design errors in combinational logic introduce an additional risk for vulnerabilities to fault attacks.

Physical fault effects are required to be abstracted using behavioral models, so called fault models, when using fault modeling tools to mimic faults in circuits. This way, the degree of abstraction is adapted to the use case in order to reduce fault modeling complexity. This increases the efficiency of fault modeling tools, which is especially of importance when huge sets of different faults need to be mimicked to generate significant results. Fault models map complex physical behavior to temporal, spatial and value properties such as timing, fault duration, fault location, multiplicity (spatial and temporal), type and value given by the fault model type. These properties are implemented depending on the target abstraction level, which can be transistor level (electrical), gate level (logical), register-transfer level (functional), algorithm level, software layer and system level (architectural, including hardware and software). In Figure 3.3 configuration possibilities of fault models are summarized, which are discussed subsequently.

3.3.1 Abstraction Level of Fault Models

Fault models can be divided roughly into two classes depending on the supported precision of the time granularity and the level of abstraction to which these are applied. The first class includes cycle accurate fault models used to model digital behavior at zero delay gate level and higher abstraction levels that abstract timing details and do not consider electrical and latching-window masking (detailed in Section 3.5). Although, the level of abstraction is relatively low in comparison to possible levels of abstraction, these fault models are often named *high-level* fault model in literature, e.g. in [PHRB11]. The second class contains fault models that aim on accurate modeling of physical effects, for which a more precise time granularity is required and technology-dependent properties of circuits are considered for fault modeling, especially such affecting fault propagation of transient faults. For this purpose, low-level details such as the used manufacturing technology, propagation delays and attenuation of transient pulses in combinational logic are taken into account. These are applied to back-annotated gate level and lower abstraction levels such as transistor level, full custom and analog circuits and are often considered a *low-level* fault model in literature, e.g. in [PHRB11]. These fault models are usually applied when it is of interest to study physical effects of faults in great detail, e.g. for basic research, and also when the Soft-Error-Rate (SER) for reliability engineering is determined.

In this thesis, fault models are used to mimic fault effects resulting from fault attacks, which results in the following dilemma: On the one hand, for precisely mimicking fault effects caused by arbitrary fault attacks, fault models at lower levels of abstraction are required. This requires a lot of computational effort, and effort is often spent on cases where faults are masked and do not result in errors. For instance, faults modeled in transistors of a combinational cell at transistor level may not effect the output of the respective gate. On the other hand, the interests are in modeling fault effects at higher abstraction levels that allow to keep the complexity of fault injection and fault propagation in check, required for exhaustive analysis of the behavior of security circuits in presence of faults [PHB$^+$14]. For this purpose, technology independent models are usually used that are available early during circuit design when details such as circuit timing are not available [PHB$^+$14, PHRB11]. Furthermore, cases should be focused where faults are not masked and actually lead to errors, i.e. the worst case scenario, which is of importance for validating fault countermeasures [PHRB11] and for identifying security flaws early during circuit design [PTH$^+$15].

To ensure applicability for security verification, the following descriptions will mainly consult cycle accurate fault models suitable at gate level and register-transfer level, which basically constitutes a trade-off between precision and performance. This enables fault modeling in combinational and sequential cells with a level of abstraction that allows to mimic arbitrary fault attacks. Huge sets of different faults representing relevant situations with respect to physical fault injection in digital logic can be analyzed, where limited verification time is used efficiently.

3.3.2 Properties of Fault Models

Properties of physical faults are mapped to properties of fault models such as discrete locations and timing with a granularity that depends on the circuit's level of abstraction under consideration.

Spatial Properties The spatial property of fault models configures the locations for which faults are to be modeled, where the circuit's level of abstraction has to be taken into account. Often the term spatial fault multiplicity is used to describe the number of faulty locations, which helps to generalize spatial properties for single and multiple fault injection.

Temporal Properties Temporal properties of fault models include the fault injection time and the fault release time relative to the circuit's current state, which depends on the circuit's input or software execution. Furthermore, the fault duration describes the time frame in the bounds of the fault injection time and the fault release time. As far as modeling faults for multiple time intervals, which do not have to be consecutive time intervals, is concerned, the term temporal fault multiplicity (see e.g. [Lev05]) can be used to describe the number of affected time intervals. Although the temporal fault multiplicity helps to generalize temporal properties, it is not sufficient when modeling higher order fault attacks, e.g., multiple time-displaced attacks, since the information about the fault injection and release times is not considered. This aspect is further discussed in detail along with formal descriptions in Chapter 4.

Considering these temporal properties is especially important in the security context to describe effects of e.g. lasers, which are able to generate a sequence of pulses, injecting several faults in a short time [Lev07]. This was exploited to mount a second order fault attack in [TK10], which successfully broke a protected software implementation of a CRT-RSA cryptographic algorithm.

Value Properties Value properties represent the fault injection. Both, the value and the spatial properties, are tightly coupled with the level of abstraction under consideration. For instance, it does not make sense to model a byte fault on transistors, whereas a bit-flip can be modeled at several levels of abstraction including gate level, RT level and software layer.

3.3.3 Fault Model Types

Analogously to faults in integrated circuits, as discussed in Section 2.2.3, fault models are classified based on the persistence of to be modeled faults into intermittent, permanent and transient fault models. Moreover, a fault model type known from literature defines a specific configuration of spatial, value and temporal properties. Furthermore, a fault model type maps fault effects to a value space and spatial configuration space dependent on the abstraction level under consideration. Therefore, fault model types reported in literature are usually already tailored to fit a certain level of abstraction as well.

In a security context, transient fault models are preferred since these are able to mimic the timing precision required to break modern hardened security and cryptographic devices. There have been attempts to use permanent fault models as well, mainly to mimic faults in non-volatile memory. For instance, the first model for differential fault attacks on secret key cryptographic systems [BS97] was based on a permanent fault model.

Therefore, I am going to focus on transient and permanent fault models for the remainder of this thesis. Intermittent faults are not relevant for this thesis and are not further considered.

3.3.4 Transient Fault Models

Providing configurable temporal properties, namely fault injection time and fault release time, in addition to spatial fault properties is characteristic for transient fault models. This allows to model single and multiple transients in combinational logic (SET, MET) and single and multiple bit-upsets in memory elements (SEU, MEU). Sometimes, the fault duration or a temporal fault multiplicity is considered as well. The former enables to model faults for consecutive time intervals, whereas the letter helps to generalize descriptions for faults affecting consecutive and non-consecutive time intervals. Therefore, transient fault models suit to mimic faults induced by fault attacks, as will be further detailed subsequently.

For mimicking physical fault attacks in a security context, the set/reset and bit-flip fault models are suitable, as stated in literature for e.g. laser-induced faults [PHB$^+$14, VML$^+$14, RSDT13], glitching attacks [ADN$^+$10] and EM glitch attacks [OGST$^+$14]. The bit-flip fault model mimics faulty behavior by inverting the value of a fault site, e.g. the stored value of a memory element. Set and reset fault models, however, mimic faulty behavior by forcing either a logical '0' or logical '1' regardless of the fault-free value. Compared to set and reset fault models, the bit-flip fault model is more pessimistic in the sense that whenever a fault is injected, it actually causes the fault site to change its value. In contrast, a reset fault e.g. modeled for one clock cycle for a sequential cell that already drives logical '0' would not cause any error. Set and reset fault models mimic fault behavior in a more realistic fashion [SBHS15, PHB$^+$14, VML$^+$14, RSDT13], especially when considering multiple faults. However, since the bit-flip model mimics the worst case [VML$^+$14] it is more efficient for the purpose of evaluating fault countermeasures.

Usually, the bit-flip fault model is applied to sequential cells and memory elements, for which it is applied once at the desired point in time. After applying the bit-flip fault model, the faulty value is effective until it is overwritten functionally by the circuit. When applying the bit-flip fault model to combinational cells, it is important to make sure that the faulty value either does not toggle multiple times dependent on the switching activity of the combinational logic or is captured at most once by downstreaming sequential cells or memory elements. Otherwise, actual physical fault behavior would not be modeled in a realistic way. This is detailed later in Sections 6.1.2 and 6.1.3, where I present a fault model that combines the pessimistic characteristic of the bit-flip fault model with the ability to hold the faulty value for an arbitrary duration.

Note that the ability to configure a temporal fault multiplicity with varying spatial fault properties would enable to mimic multiple fault injection times, which would allow to generalize fault models. For example, this would allow the bit-flip fault model to mimic the fault effects covered by multiple set and reset faults as well.

3.3.5 Permanent Fault Models

Mimicking permanent faults have been established for decades in EDA tools such as fault simulators and automated test pattern generation (ATPG) tools. Characteristic for permanent fault models is that these have only one valid configuration of temporal properties: the fault is injected at the very beginning of a test and it is released at the very end of a test. Therefore, the number of possible fault injection times is exactly one and the temporal fault multiplicity is given by the test length. Permanent fault models that can be applied at gate level include stuck-at fault models, path fault models, transition fault models and bridging fault models. These are usually used to mimic single or multiple manufacturing defects in combinational or sequential cells for e.g. test pattern generation, fault simulation and on-line testing. For modeling permanent faults at transistor level, the stuck-open and stuck-on fault models are usually used. The following description of permanent fault models is mainly inline with [ABF94] and [KKJ10].

In case of stuck-at fault models applied at gate level, a fault is represented by a faulty gate whose output is permanently forced to either logical '0' or logical '1', affecting the connected net as well. The same concept can also be applied to gate inputs, however, then the associated net is considered disconnected, thus, the fault is not propagating into it. This mimics the fault effect of a defect input or output transistor that permanently closes or opens, respectively. Based on the spatial fault multiplicity of faults, stuck-at fault models are further divided into single stuck-at and multiple stuck-at fault models [KKJ10]. The stuck-open and stuck-on fault models are corresponding models at transistor level.

Bridging fault models mimic an electrical short between two nets (interconnect bridge), where the short is modeled as either wired-OR or wired-AND, such that one driver dominates the other. In this way, the dominating value is determined, which is then assigned to both nets.

Delay fault models such as transition and path fault models mimic manufacturing defects that causes gates or entire combinational paths to react more slowly than specified, e.g. caused by resistive shorts or process variation. The transition fault model assumes that the defect causes any signal transition to be delayed past the clock edge. In contrast, the path fault model assumes a distributed delay along a combinational path.

3.3.6 Summary and Conclusion

Fault models usually describe the specific and small subset of the fault injection space that is relevant to a particular application domain in which it is originated. This can be shown with e.g. the stuck-at fault model, where even single stuck-at faults and multiple stuck-at faults are separated into two dedicated fault models [KKJ10]. The relationship between single and multiple faults could also be described in a more general way when considering the spatial multiplicity of faults, i.e. the number of affected locations. This way, single stuck-at faults just constitute the small subset of multiple stuck-at faults for which the spatial fault multiplicity is exactly one.

The same consideration can also be applied to generalize temporal properties by introducing a temporal fault multiplicity. Moreover, introducing the number of fault injection times would help to describe the number of events that are required to be handled by fault modeling tools when multiple fault injection times are considered.

Furthermore, Fault models are usually bound to a specific abstraction level. Sometimes fault modeling even entangles both fault injection and modeling of circuit behavior. This can be noticed when looking at fault models that consider technology-dependent attenuation of transient pulses and circuit timing.

These considerations motivated me to introduce a fault configuration model at a meta level, presented in the next Chapter 4, which is generalized in order to be independent of specific abstraction levels and superior to specific fault models. It covers the entire fault injection space including arbitrary fault configurations w.r.t. spatial, value and temporal properties. Moreover, both the spatial fault multiplicity and the temporal fault multiplicity as well as multiple fault injection times are considered in the model. The meta fault configuration model can be further refined for the abstraction level under consideration, which I show by means of specifying and implementing a fault configuration model at gate level in Chapter 6. To use the same fault model to mimic all common physical fault attacks, it needs to be as general as possible [VML$^+$14]. Therefore, the presented meta fault configuration model features single and multiple transient faults as well as permanent faults in combinational and sequential logic uniformly, and hence, suits modeling arbitrary fault attacks.

3.4 Testing Concepts

When the workload of a circuit is designed to explicitly sensitize certain circuit parts, e.g. to check proper functionality, it is also referred to as a test in the context of fault modeling. Parts of a circuit that are not sensitized may remain idle, i.e. signal transitions do not occur in these circuit parts. This can be the reason for a fault to remain in a system as an error without causing a service failure in external states observable at primary outputs. Moreover, a fault may also disappear from the system, when it stops causing errors at some point in time. As for the purpose of manufacturing testing and on-line testing, efficient tests that are able to make hardware faults observable as errors in internal states or as service failures are therefore required for security verification of fault countermeasures. Furthermore, in a security context, a test is used as representative of relevant situations and is required to make use of to be verified hardware components by performing, e.g., encryption and decryption. Subsequently, testing concepts and the related nomenclature are briefly introduced.

3.4.1 Testability of Faults in Combinational Logic

Figure 3.4 adds the conditions sensitization and propagation as well as the property observability to the fault-error-failure chain, which was depicted earlier. Related definitions are given subsequently, which are mainly inline with [ABF94]. These were originated from manufacturing testing and were originally introduced for permanent faults in combinational circuits. Further concepts have been introduced to increase testability for sequential circuits, which are also outlined very briefly.

Sensitization and Activation A logic line, i.e. a circuit node, is said to be sensitized to a fault by a test (e.g. input pattern on primary inputs of a circuit), if it drives the negation of the fault-free value in the presence of the fault. A path of sensitized logic lines is referred to as a sensitized path. For instance, to sensitize a fault that shorts a logic line to ground, modeled as stuck-at-0 fault, the logic line has to drive a logical '1' in the fault-free case. This is also referred to as activation in literature. Hereinafter, I use the term sensitization exclusively. My intention is to avoid confusion between the terms fault activation and fault injection later on.

Fault and Error Propagation Fault propagation and error propagation are often interchangeable used in the literature. Propagation occurs if other locations are also erroneously affected by the initial fault or by the caused error.

Observability Observability is the ability to observe an internal logic line directly at primary outputs (PO). That is, if at least one primary output switches depending on the to be observed internal logic line, then the respective logic line is observable. Furthermore, if a fault propagates to at least one PO, where it is observable, then the fault is said to be detected. In contrast, if it is not observable, then it is said to be undetected. In the context of manufacturing testing, a detected fault constitutes a service failure.

Controllability Controllability is the ability to establish specific values at logic lines in a circuit by applying a stimulus at primary inputs (PI). Controllability is established if a test applied at primary inputs (PI) sensitizes a path from PIs to the fault site and a path from the fault site to primary outputs POs, such that a fault propagates to POs where it is observable. In case that no sensitized path exists, the fault is said to be logically masked. Controllability is very similar to observability, however, with the difference that controllability describes the ability to control logic values of logic lines by applying a stimulus, whereas observability describes the ability to determine signal values of logic lines at POs by applying a stimulus at PIs.

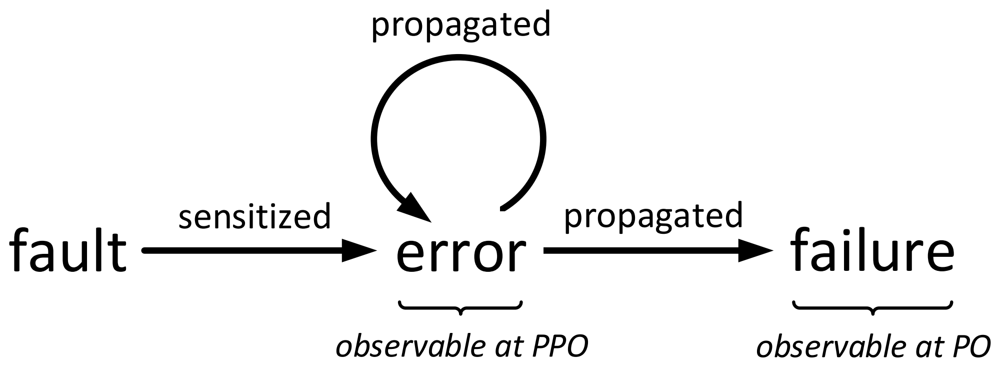


Figure 3.4: Sensitization, propagation and observation illustrated along with the fault-error-failure chain.

Testability Testability is the ability to control a fault with a test and to observe it at POs.

3.4.2 Testing Concepts for Sequential Circuits

Hennie [Hen61] introduced the iterative logic array (ILA) model, which enables to also apply the testing concepts of combinational circuits to sequential circuits. Basically, this model unrolls sequential circuits for a given number of cycles by means of concatenating a corresponding number of copies of the combinational circuit part. In order to increase the controllability and observability and in turn the testability for sequential circuits, the concept of pseudo primary inputs (PPI) and pseudo primary outputs (PPO) was introduced. These ports are the inputs and outputs of the combinational circuit part and are connected to either primary ports or sequential elements (memory and flip flops). This way, tests are applied at PPIs and faults can be observed at PPOs, i.e. the testability of sequential circuits is increased. This concept is nowadays routinely applied when considering design for testability (DFT) techniques, utilizing e.g. scan register chains to control PPIs and to observe PPOs. For this, registers are connected to chains used to transport a test from dedicated PIs to PPI and to transport the combinatorial response from PPOs to dedicated POs, which is activated in a dedicated test mode.

3.4.3 Tests and Test Generation

Note that the problem of test generation is out of scope of this thesis. However, in order to comprehensively understand the concept of fault injection and respective tools, related concepts are outlined briefly from a general point of view. Later in Section 5.4, I review how tests are generated, uploaded and executed when using fault emulation. The objectives on functional tests within the scope of verifying fault countermeasures are detailed in Section 6.5.4.

Test generation is a complex and time consuming problem. The goal of test generation is optimizing test sequences in order to maximize test or fault coverage while minimizing the test duration. Tests are subdivided into two classes: structural tests and functional tests.

Structural Tests Structural tests are stimulus (singular), stimuli (plural) or input pattern, which are applied at PIs of combinational circuits. In case of sequential circuits, FFs can be connected to scan register chains to increase testability. Structural test can be generated based on the structure of the circuit under test, i.e. independently of its actual functionality, using commercial automated test pattern generation (ATPG) tools. ATPG tools are optimized to find single test pattern that already detect a set of faults and to find compact sets of tests that cover, if possible, all faults.

Functional Tests Functional tests include software-based tests for processor architectures fetched from memory. Functional tests are used when structural testing of processors is technically or economically infeasible [CMD01] due to limited top-level interfaces, which render sensitization and observation impracticable. Test software can be a specific workload or target application. For example, Mohammadi et al. [MEEM12] use different sorting algorithms as workload to sensitize a processor architecture. There is also a class of software that is especially built for test purposes, which is referred to as software-based self-test (SBST) [CMD01, PGSR10]. In order to make errors in internal states observable, test results can be propagated to observable locations, e.g. addressable memory [Reo15].

In contrast to structural tests, commercial tool-support is not available for automated test generation of functional tests [Reo15]. Thus, functional tests are written manually by a test engineer. Recently, there are attempts to automate functional test generation in scientific communities as well as industry [Reo15]. Riefert et al. [RCS$^+$15] present the first method able to automatically generate functional test programs with superior fault efficiency compared to those produced with manual approaches. However, the authors state that high computational effort is required, constraints disabling external interrupts are applied and the method is so far applicable for permanent stuck-at faults only.

3.5 Propagation and Classification of Transient Faults

Fault propagation of static permanent faults can be masked only by logical fault masking (see also the description of controllability in Section 3.4). In contrast, dynamic permanent faults can be masked by latching window masking as well, which is the reason why the behavior of these faults depends on the clock frequency. Unlike permanent faults, fault propagation of transient faults is derated by in total tree different masking effects, namely, logical masking, latching window masking and electrical masking. Masking effects of transient faults are discussed next.

3.5.1 Masking Effects

All three masking effects can prevent fault propagation, and therefore, have an impact on whether a transient fault causes an error or failure. Electrical masking and latching window masking are especially relevant when modeling single and multiple event transients (SET, MET), i.e. when modeling transient faults in combinational logic.

Logical Masking Logical masking occurs when a fault site is sensitized by a test, but a sensitized path to (pseudo) primary outputs does not exist (see Section 3.4). That is, logic in the fan-out cone of the fault site does not propagate the fault from input pins to output pins. A fault might propagate to one or more input pins of the respective logic gates but the gates' outputs are logically dominated by the unaffected fault-free inputs. There is also the case of fault masking, where one fault cancels out another fault [Dia75], i.e. one fault logically masks another.

Electrical Masking Electrical masking effects are a consequence of delay degradation as a transient is propagated through combinational logic gates [EVC$^+$09]. It attenuates the pulse width of a fault while it propagates through a sensitized combinational path [LDJK94] and is caused by the input and load capacitance and the effective parasitic capacitance between input and output of the gates it passes [WX11].

Latching Window Masking (Temporal Masking) If a transient pulse arrives at downstreaming sequential cells' inputs, but not within the setup and hold time, then it is not latched by the respective sequential cells. This is

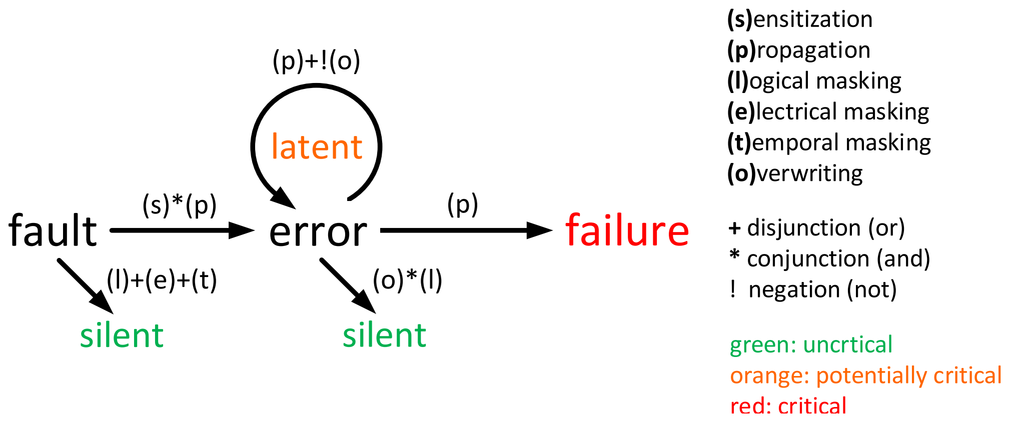


Figure 3.5: Detailed fault classification into failure and passes along with the fault-error-failure chain considering logical, electrical and temporal masking effects. Passes are further subdivided into silent and latent faults.

referred to as latching window masking [Gai97] and is also known as temporal masking in literature.

3.5.2 Detailed Fault Classification

Since faults may remain in memory elements without causing a service failure or may disappear from memory elements, passes are further subdivided in latent and silent faults. This relation is depicted in Figure 3.5 and is subsequently explained along with masking effects of transient faults. Therefore, fault propagation, fault overwriting and fault masking effects are considered in addition in Figure 3.5 (listed in its legend) to complete the causality relationship of transient faults, errors and failures. The logical conjunction $*$ of these effects indicates that multiple effects are required, whereas the logical disjunction $+$ indicates that at least one effect is required. The logical negation ! indicates that an effect must not occur.

Silent Faults A fault is classified as silent, if it disappears due to masking effects or overwriting. There are two cases for faults to exhibit a silent behavior. Firstly, the fault is never latched in sequential cells because of logical, electrical or temporal (latching window) masking, which is indicated by $(l) + (e) + (t)$ in Figure 3.5. Secondly, the faulty value of a memory element is overwritten before it is latched by other memory elements while logical masking prevents a further fault propagation into other sequential cells, indicated by $(o) * (l)$ in Figure 3.5.

Example 3.5.1 (Silent fault)**.** To give an example, a transient fault such as a SEU in a memory element of a processor design has the potential to result in a service failure, where e.g. the software execution of the processor crashes. For this to happen, the fault has to be injected or propagated into a memory element that is read by other hardware components (could be initiated from software). However, if this location is overwritten by either hardware or software before being read again, then the fault would disappear, resulting in a silent fault.

Latent Faults A fault is classified as latent, if it manifests as an error in the internal state (e.g. memory elements), where it is still present at the end of a test but does not cause a system failure. There are two cases that may lead to this behavior, which are depicted as transition from error to error and indicated by (p) in Figure 3.5. Firstly, the fault remains in exactly the memory elements where it was initially injected or propagated into from combinational logic. Secondly, the fault propagates to other memory elements, but still does not cause a system failure.

Example 3.5.2 (Latent fault)**.** When a erroneous memory element is read, it could cause a service failure immediately or it could propagate to other memory elements and then as a result cause a service failure. In contrast, if the fault remains in the internal state but does not cause a service failure, then it is said to be a latent fault. This might happen if e.g. at some point the erroneous memory elements are not accessed anymore, neither reading nor writing. More general speaking, when the external state does not depend on the erroneous memory elements to which the fault propagates, then a service failure cannot occur.

3.6 Pre-Silicon Fault Modeling Tools

Security verification methods are subdivided into pre-silicon and post-silicon methods. In contrast to physical fault injection techniques, which belong to post-silicon methods (refer to Section 2.3.2), pre-silicon methods make use of fault models to mimic faults and are hence applicable during circuit design. In general, it is desirable to find design bugs early in the design flow of a circuit. When circuit design advances, fixing design bugs becomes more expensive and delays time-to-market. Therefore, pre-silicon methods available early during circuit design are required for security verification of fault countermeasures, where fault models are used to mimic fault attacks. Another motivation for pre-silicon methods is to have tools which can be used to track security issues encountered during post-silicon security evaluations.

There are different methodologies for fault modeling reported in the literature:

- Logic simulation
- Fault simulation
- Formal methods
- Analytical Soft-Error-Rate (SER) estimation methods
- Fault emulation

Logic simulation, fault simulation and fault emulation can be used to realize exhaustive fault injection methodologies, which is usually applied during security verification. Formal methods can be used for security verification of fault countermeasures implemented in hardware only to a limited extend because of complexity issues. Analytical SER estimation methods are typically used in the application domain of reliability engineering.

Subsequently, I briefly recap these pre-silicon fault modeling methods along with the literature and place them in the application domains where they were originated and usually used. The focus lies on emphasizing methods applicable for security verification of fault countermeasures implemented in hardware.

Logic Simulation Simulation setups are usually used for functional verification and are, thus, usually already available in development environments. Two different approaches using logic simulation for fault injection are reported in the literature using either built-in simulation commands [JAR$^+$94] or mutants and saboteurs based on HDL modification [GC91, JAR$^+$94].

Built-in simulation commands can be used to inject and release faults in a circuit during simulation. This is the easiest to implement method, since it does not require HDL modification, and it works at abstraction levels ranging from transistor level (electrical level) to system level (architectural level).

Mutants and saboteurs enable fault injection capability by HDL modification. Mutants replace the original component register transfer level (RTL) description by a description that is capable of fault injection. Saboteurs are capable of altering signal behavior and extend an RTL design without replacing the original description [JAR$^+$94]. There are also attempts at higher abstraction levels. For instance, Misera et al. [MVS08] introduced mutants and saboteurs to SystemC-based architectural simulation. In general, simulation-based fault injection techniques are considered to be slow such that exhaustive fault injection campaigns cannot be realized efficiently using slow simulations.

Fault Simulation In order to increase simulation performance in case that faults are considered to be present in the circuit, dedicated fault simulation was introduced. Fault simulation was originated in the manufacturing testing domain. Several performance optimizations such as fault dropping and parallel, deductive, differential and concurrent fault simulation have been proposed, allowing fault simulation to perform noticeable faster than exploiting logic simulation for fault injection. Commercial fault simulators, e.g. WinterLogic Z01X and Cadence Verifault-XL, provided only permanent fault models for a long time since transient faults are irrelevant for manufacturing testing. Similarly, academic fault simulators, e.g. HOPE [LH92] and PROOFS [NtCP92], provide only permanent stuck-at fault models. Ever since automotive domains required safety verification considering standards like ISO 26262, transient fault models were supported by commercial fault simulators. This is still an emerging topic taking into account that there is not yet a single commercial fault simulator available that supports fault injection of multiple transient faults in combinational logic.

Motivated from a security context, a simulation-based approach (named tLIFTING) that mimics multiple event transients at transistor level (SPICE simulation) and maps these to higher abstraction levels (logic simulation) was presented by Bosio et al. [DNFLR12]. tLIFTING extends an earlier published fault and logic simulator (LIFTING [BDN08]) which is able to simulate single and multiple stuck-at faults as well as single event upsets. Although this simulator is able to inject faults, strictly speaking, LIFTING is not a dedicated fault simulator since it does not come with performance optimizations that are typical for fault simulation.

In case the CUV is a processor-based architecture, processor simulation setups such as for example [TMS$^+$13] can be used to inject faults into CUV's addressable registers. Since fault injection is restricted to addressable registers, these setups are only suitable for software verification.

Formal Methods Formal methods can be used to verify that hardware, software or algorithms comply with their specification using equivalence checking, model checking or theorem proving. In cases where a proof fails, a counter example is provided. Nowadays, automated test pattern generation (ATPG) is also supported by using formal engines, such as e.g. SAT-solvers (Boolean Satisfiability Problem-solver). Formal methods are very expensive in terms of computational effort, which increases drastically with the state space to be explored. The complexity of hardware verification usually renders formal verification of an entire system already impossible without considering fault modeling. Hardware verification therefore usually applies functional verification using logic simulation.

Nevertheless, there have been attempts to adapt formal methods for fault modeling. For instance, Larsson et al. [LH07] and Pattabiraman et al. [PNKI13] present frameworks for symbolic fault injection of SEUs applied to a Cyclic Redundancy Check (CRC) algorithm and an aircraft collision avoidance application, respectively. Both frameworks model fault injection, where faults are explicitly enumerated, similar to simulation- and emulation-based exhaustive methods. Furthermore, fault injection is only possible at software layer (fault injection in addressable registers). Both methods are therefore not suitable for hardware verification.

Contrarily to exhaustive approaches, formal methods can approach the problem the other way around. Instead of injecting different faults into a circuit and observe the circuit's behavior in an exhaustive fashion, formal methods can represent faults in a more general way. For this, properties are used to express that the circuit is in a faulty state, where assumptions can be made to restrict the multiplicity of multiple faults. Such approaches, used for dependability analysis, were presented by e.g. Leveugle [Lev05] and Baarir et al. [BBC$^+$09], but results were discussed only for simple example circuits and have not been deployed yet to realistic security circuits.

Beside others, formal methods utilize efficient engines for exploring a search space. One prominent engine are SAT-solvers. SAT is the abbreviation for the Boolean satisfiability problem, which refers to solving Boolean functions and which belongs to the class of np-complete problems. A SAT-solver is capable of making decisions, such as guessing value assignments for Boolean variables. Boolean constraint propagation is then used to explore the search space until a conflict occurs or a solution is found. If conflicts occur, decisions made earlier are reversed in order to fix conflicts and to explore other parts of the search space. In this way, one solution that satisfies the Boolean function may be found, although the entire np-complete problem is not solved. SAT solvers have also been established successfully for automated test pattern generation (ATPG). Becker et al. [BDES14] provide a very comprehensive survey on recent advances in this regard.

Analytical SER Estimation Methods Analytical Soft-Error-Rate (SER) estimation methods stem from reliability engineering domains and include statistical, analytical, probabilistic and symbolic techniques, such as presented by Miskov-Zivanov [MZM10], Polian et al. [PHRB11] and Chen et al. [CET13]. Contrarily to fault injection approaches, these methods do not approach the problem of fault modeling exhaustively. Instead, the probability of a high-energetic particle strike causing faults is estimated. Usually, these methods are used to determine flip flops or combinational cells that contribute the most to the soft-error-rate. By hardening these cells selectively, the soft-error-rate can be reduced until it drops under a specific value.

Analytical SER estimation methods are not sufficient to validate the effectiveness of fault countermeasures in a security context since fault probabilities are estimated, which only suits modeling random fault occurrence. In a security context, however, faults are deliberately, precisely and locally injected, e.g. using laser beams.

Fault Emulation Cheng et al. [CHD95] proposed to use FPGA-based emulation as accelerator for fault grading of manufacturing tests. Since then, FPGA-based fault emulation has also been used to implement fault injection campaigns in other application domains, namely reliability, safety-critical and security-critical engineering.

Fault emulation environments alter a CUV utilizing for example the circuit instrumentation technique [EMEM14, EVC+09, KLPB05] to provide fault injection capability. The instrumented CUV is then synthesized onto an FPGA. Alternatively, commercial hardware prototyping platforms such as Cadence Palladium maybe utilized, as presented by e.g. Daveau et al. [DBG+09].

FPGA-based fault emulation is the fasted method to perform extensive fault injection campaigns. It is three to five orders of magnitude faster [EMEM14, EVC+09] than simulation and other software-based statistical, probabilistic and symbolic approaches. This high performance provided by fault emulation allows to benefit as much as possible from limited verification times during circuit design, which constitutes the reason for choosing FPGA-based fault emulation as fault injection method for security verification in this thesis.

In Section 5 FPGA-based fault emulation is detailed in an own section along with the related work. The implementation of a specific FPGA-based fault emulation environment, which serves as basis for further performance optimizations and features, is presented in Chapter 6.

3.7 Relevance for this Thesis

In the remainder of this thesis, fault emulation is chosen as central method for fault modeling used to implement fault injection campaigns during security verification, which is detailed along with the literature in Chapter 5. A particular implementation is presented in Chapter 6. The high performance provided by fault emulation is independent of the CUV's circuit size and allows to use limited time for verification efficiently early during circuit design. Furthermore, long functional tests required to sensitize processor designs are applicable. Of course emulation-based techniques also have some disadvantages and limitations. One of the main contribution of this thesis is to remedy these.

There are for example limited hardware resources on FPGAs, which in turn limits the circuit size for which fault emulation is applicable, especially when fault injection in combinational logic is considered. To tackle this issue, in Chapter 7, I introduce a software-based pre-processing for faults in combinational logic that maps faults in combinational logic to equivalent faults in sequential logic, which are configured onto a fault emulator. For this purpose, I additionally make use of software-based fault modeling techniques based on Boolean constraint propagation provided by a SAT-solver.

Another disadvantage of fault emulation tackled in this thesis is that performance optimizations often come at the cost of less flexibility in terms of controllability. I tackle this issue by techniques that are generic in the sense that these allow arbitrary fault configurations in terms of spatial and temporal properties. This high configurability is supported by performance optimizations presented in Chapter 8, which closes the gab between speed and configurability of fault emulation environments.

In order to provide formal descriptions for the presented techniques and implementations, I first present a meta fault configuration model in the next Chapter 4. Furthermore, Chapter 4 aims on creating an understanding of fault injection complexity in general and provides equations that allow to determine the fault injection complexity for practice-oriented subsets of the fault configuration space.

Chapter 4

Meta Fault Configuration Model

After detailing fault effects and reviewing fault modeling concepts in previous chapters, this chapter provides the basis for formal descriptions of the following fault injection techniques and respective implementations. Moreover, my intention in this chapter is to create a comprehensive understanding about mapping properties of physical faults onto fault configurations using fault models and about the resulting fault injection complexity when considering configuration possibilities that are required to mimic arbitrary faults and fault attacks. For this purpose, I define a fault configuration model that scales in terms of spatial, value and temporal granularity for physical fault attacks ranging from less precise attacks (e.g. glitching attacks) to precise laser attacks. Since fault modeling can be applied at any level of abstraction of a circuit under verification, I decided to formulate the presented fault configuration model independently of particular abstraction levels, i.e. it is defined at a *meta* level. This way, its fault properties can be refined in order to apply the meta fault configuration model to the abstraction level under consideration, maximizing its applicability. Another very important thing to note is that the presented model solely covers fault configuration to describe fault injection in a system, but it is not meant to model the behavior of the system. To emphasize on this matter, I decided to name it a *fault configuration model.*

The meta fault configuration model describes fault configuration functions defined in a function space, the total configuration space, which allows to describe all configuration possibilities of fault models (illustrated earlier in Figure 3.3). Thus, all configuration possibilities that need to be considered for mimicking arbitrary fault attacks are covered. The meta fault configuration model therefore is a superset of all fault models known from literature. Note that also the temporal spread of faults, defined as temporal fault multiplicity, and arbitrary multiple fault injection times are covered. For this it is required that arbitrary spatial fault configurations can be defined for arbitrary time intervals of a fault experiment. Therefore, the meta fault configuration model describes the time line of a fault experiment with consecutive time intervals. For each time interval an arbitrary fault configuration, which covers spatial and value fault properties, can be applied, resulting in a sequence of fault configurations. In summary, every fault configuration defined in the total configuration space describes a distinct sequence of fault configurations, representing a particular fault experiment, and the total configuration space covers all possible configuration sequences.

In Section 4.1, I develop the meta fault configuration model based on its three components, namely temporal properties, spatial configuration space and value configuration space, which I join into a function space, the total configuration space. Then, in Section 4.2, the terms spatial fault multiplicity and temporal fault multiplicity are formulated in the context of this model, where single and multiple affected locations and single and multiple affected time intervals are discussed. In Section 4.3 a practice-oriented interpretation of the presented fault configuration model is discussed, where terms commonly used in literature to describe temporal properties of faults such as single and multiple fault injection times and permanent and transient fault models are formulated in the meta fault configuration model and discussed accordingly. Moreover, the formal description of a particular parametrized fault F is introduced, which bridges the gap to the description used in literature to describe particular faults when for example fault testing is concerned. Finally, I discuss the fault injection complexity of fault injection campaigns in Section 4.4 to create a comprehensive understanding in this regard. I also present the fault injection complexity of practice-oriented and meaningful subsets. This is supported by equations that allow to parametrize the fault injection complexity with the spatial fault multiplicity and the number of fault injection times (the temporal fault multiplicity can be used alternatively). The developed equations constitute a powerful and practice-oriented tool to determine the complexity of sophisticated fault injection campaigns.

A fault configuration model for faults in sequential and combinational logic at gate level is derived later in Chapter 6.1 by means of refining the properties of the meta fault configuration model for gate level. This way, a fault configuration model at gate level is derived, which is used to describe the presented fault injection techniques that are implemented using FPGA-based fault emulation in Chapter 6 and extended by a software-based method in Chapter 7.

4.1 Fault Configuration Space

A physical fault has spatial, value and temporal properties. With respect to fault models, properties of physical faults are mapped onto discrete fault locations and discrete timing, for which also the spatial spread and temporal spread of faults need to be considered. This way, arbitrary fault attacks can be mimicked, including more sophisticated high-order fault attacks, e.g. second-order fault attacks that may cause two different faults from a single source or from two independent sources activated at different times.

The granularity, with which these properties are mapped, depends on the level of detail that the fault model is able to resolve and the abstraction level under consideration. For example, the spatial granularity of fault models at transistor level allows to map faults onto transistors, which cannot be implemented using fault models that are tailored to higher levels of abstraction. A higher temporal granularity allows to model shorter transient faults, e.g. a transient voltage pulse for a duration of a fraction of a clock period, which cannot be implemented using cycle accurate fault models. There is also the type of a fault model, which is represented by discrete values depending on the level of abstraction under consideration. At higher abstraction levels the type is usually represented by only a few possibilities, e.g. stuck-at-0 and stuck-at-1, and therefore, is often payed little attention when modeling faults in digital circuits. However, when combining several fault model types (bit-flip, stuck-at, etc.) into a single implementation, as I am going to do in Chapter 6, it becomes necessary to describe these in a single value space in order to be able to treat different fault model types uniformly. Moreover, considering fault models that are applicable to analog circuits, the value granularity becomes increasingly important. As a consequence, the spatial and value configuration spaces as well as the temporal granularity of a meta fault configuration model are required to be independent of specific levels of abstraction. Furthermore, in order to derive fault configuration models for specific levels of abstraction based on the meta fault configuration model, it is also necessary to build the model in a way that it allows a refinement of its properties.

With these considerations and requirements in mind, I am going to define the fault configuration space of the meta fault configuration model, starting with the temporal granularity.

4.1.1 Temporal Granularity

Temporal properties of faults describe when and how long these faults occur. The temporal granularity, with which the temporal properties are resolved, is defined next on the base of consecutive time intervals on which the test duration of a fault experiment is mapped. The bounds of these time intervals are defined by discrete instances in time, hereinafter denoted by t_i, where $t_i \in \mathbb{R}$ is the index assigned to identify the time interval and $t_i < t_{i+1} \leq t_{N_T}$ and $0 \leq i < N_T$. Using time intervals instead of mapping temporal properties directly to discrete instances in time suits the purpose of providing a formal description for following implementations of fault injection methods better. Important to note is that this description enables to model the fault configuration for an entire fault experiment, where for each time interval an arbitrary fault configuration with arbitrary spatial and value properties can be modeled. That is, multiple fault injection times during fault experiments can be modeled, which is required to model physical faults mounted by a single source in a time-displaced manner or mounted by multiple sources. Thus, arbitrary physical fault attacks including higher order attacks can be modeled.

Definition 4.1.1 (Time Intervals)**.** Fault injection is resolved for a time interval, denoted by $T_i = [t_i, t_{i+1})$, given by two consecutive discrete times $t_i, t_{i+1} \in \mathbb{R}$, where $i \in \mathbb{N}_0$ and $0 \leq i < N_T$. In each time interval an arbitrary spatial fault configuration and value configuration can be applied, which are defined in respective configuration spaces in the following sections. The set T includes all time intervals to which a fault experiment is mapped, so $T_i \in T$ where

$$T = \{\, T_i \mid T_i = [t_i, t_{i+1}),\ T_i \subset \mathbb{R},\ t_i < t_{i+1} \leq t_{N_T},\ 0 \leq i < N_T \,\}\,. \tag{4.1}$$

Next, the temporal granularity, with which temporal properties are resolved, is defined.

Definition 4.1.2 (Temporal Granularity)**.** The cardinality of T is the total number of time intervals, which corresponds to the temporal granularity of a fault model, denoted by N_T, where $N_T = |T|$.

Note that $N_T + 1$ discrete instances of time are required to define N_T time intervals. In cases where a coarser granularity is sufficient for T, a subset of $\mathbb{R}$ can be chosen, e.g. $\mathbb{N}_0$.

Since in each time interval arbitrary spatial and value configurations can be applied, physical faults, which might be affective for multiple consecutive or non-consecutive time intervals, are modeled by assigning a sequence of spatial and value configurations. This is detailed in Sections 4.1.5 after introducing the spatial configuration space and the value configuration space.

4.1.2 Spatial Configuration Space

Subsequently, fault injection locations are introduced, based on which the spatial property of fault configurations are defined in the context of the meta fault configuration model. The corresponding definitions constitute the base for defining the spatial configuration space and its respective spatial granularity. The spatial configuration space includes fault configurations for arbitrary single and arbitrary multiple faults, which is detailed later in Section 4.2.1.

Definition 4.1.3 (Fault Injection Locations)**.** A particular discrete fault injection location, i.e. a location at which fault injection can be configured, is denoted by l_j, where $j \in \mathbb{N}_0$ is the index assigned to identify the fault injection location. The set of all considered fault injection locations is denoted by L, where $L = \{\, l_0, \dots, l_{N_L-1} \,\}$, and $0 \leq j \leq N_L - 1$.

Fault injection locations are required to be refined according to the abstraction level under consideration. For example, it can be defined as a set of coordinates, transistors, logic lines or register pins.

The spatial configuration space includes all spatial fault configurations, i.e. it includes all possibilities to combine different fault injection locations in order to model the spatial property of physical faults.

Definition 4.1.4 (Spatial Configuration Space)**.** The power set of L, denoted by $\mathcal{P}(L)$, contains all configuration possibilities of the spatial property, hereinafter referred to as spatial configuration space.

Note that the empty set $\emptyset \in \mathcal{P}(L)$ corresponds to fault-free cases, i.e. cases in which actually no fault occurs or when the fault cannot be mapped onto the abstraction level under consideration. This is important to note since $\mathcal{P}(L)$ allows to configure fault-free as well as arbitrary faulty cases, including single and multiple affected fault injection locations, for any given time interval $T_i = [t_i, t_{i+1})$. This way, a sequence of configurations for an entire fault experiment, including fault-free time intervals, can be defined.

The spatial granularity defines the number of considered fault injection locations and is defined as follows:

Definition 4.1.5 (Spatial Granularity)**.** The cardinality of L is the total number of all discrete fault injection locations l_j, which corresponds to the spatial granularity of a fault model, denoted by N_L. So, $N_L = |L|$.

The spatial properties of physical faults are mapped to none, single or multiple discrete fault injection locations. This way, during any given time interval $T_i = [t_i, t_{i+1})$, arbitrary fault injection locations are configurable to be affected by fault injection. For example, to model cases where a laser or a high-energetic particle strike affects multiple gates, multiple discrete fault injection locations can be configured to be affected.

In order to treat single and multiple faults uniformly, the spatial property of a fault configuration for a given time interval is defined as the set of fault injection locations to which the spatial fault properties of physical faults are mapped. That is, the spatial fault configuration includes the affected fault injection locations, defined as follows:

Definition 4.1.6 (Spatial Fault Configuration)**.** The set L_a denotes the spatial fault configuration, which models the spatial property of physical faults and constitutes the spatial property of fault configurations. In literature, the spatial property of faults is also referred to as fault site. L_a includes all fault injection locations affected by fault injection and is a subset of all fault injection locations included in L. So,

$$L_a = \{ \, l \mid l \text{ is affected by faults, } \; l \in L \, \} \,, \tag{4.2}$$

where $L_a \subseteq L$.

The type of fault injection locations (logic line, transistor, register pin, etc.) as well as an appropriate value configuration space representing the physical fault effect have to be refined w.r.t. the abstraction level to which the fault model is applied. Note, whether or not an affected location actually causes an error depends on the circuit's test and the circuit's internal and external state. These circuit-specific and test-dependent properties are not part of the presented fault configuration model.

So far, the temporal granularity and the spatial configuration space were defined. Next the value configuration space is introduced. Then, the relation between spatial and value configuration space is described as forcing function.

4.1.3 Value Configuration Space

The type of a fault model (e.g. stuck-at or bit-flip) is represented by a value configuration space, which has a granularity that depends on the level of abstraction under consideration. From the value configuration space values are chosen and mapped onto affected fault injection locations, which allows to mimic physical fault effects at arbitrary levels of abstraction. Considering fault injection tools such as fault simulation or fault emulation, this mapping is actually realized by forcing values at accessible locations. With this in mind, I describe the relation between spatial configuration space and value configuration space as forcing function, which is defined in a spatial-value configuration space. This way, every fault injection location can be assigned an arbitrary value from the defined value configuration space, resulting in a spatial-value configuration space.

Definition 4.1.7 (Value Configuration Space for Affected Locations)**.** Affected locations $l \in L_a$ are assigned a faulty value, denoted by v, representing the physical fault effects to be modeled at fault injection locations. Every $l \in L_a$ can be assigned a $v \in V$, where V denotes the value configuration space for affected fault injection locations.

In order to provide the ability that fault injection locations not included in L_a remain fault-free for arbitrary time intervals, i.e. these are unaffected by fault injection, it is necessary to define a value in the value configuration space to which unaffected fault injection locations are mapped. This way, every $l \in L$ can be configured individually to be affected or not by fault injection.

Definition 4.1.8 (Value Configuration Space)**.** The unaffected value, denoted by u, leaves fault injection locations unaffected when assigned to it. The value configuration space V_u includes faulty values $v \in V$ and the unaffected value $v = u$, so $V_u = V \cup \{u\}$ and $v \in V_u$.

Next, the value granularity is defined.

Definition 4.1.9 (Value Granularity)**.** The cardinality of V_u, so $|V_u|$, is the total number of values that can be assigned by a fault model, which corresponds to the value granularity of a fault model.

The value configuration space has to be refined for the actual level of abstraction and fault model type under consideration (e.g. analog or logical values), allowing to define fault model types that resolve physical fault effects with the required granularity. For example, in case of a stuck-at fault model at gate level, $V_u = \{\, 0, 1, u \,\}$ represents stuck-at-0, stuck-at-1 and unaffected, respectively. However, the stuck-at fault model describes temporal properties as well, which will be detailed in the next section. Note that different fault models known from literature can be joined into the value configuration space.

Furthermore, important to note is that the presented meta fault configuration model does not describe the circuit behavior, neither its fault-free nor its faulty behavior, and hence, fault propagation is not part of the model. The presented model only defines when and where faults occur, and the actual physical effects are mapped to values defined in the value configuration space. That is, the model does not describe whether or not a fault turns into an error. This way, fault configuration and circuit model are kept separated, which allows to refine the properties of the meta fault configuration model for the actual level of abstraction under consideration independently of circuit behavior.

So far, the temporal granularity, the spatial configuration space and the value configuration space were defined. Next the relation between spatial configuration space and value configuration space is described as forcing function, which is defined in the spatial-value configuration space.

4.1.4 Spatial-Value Configuration Space

Now, I introduce a forcing function φ, which represents fault injection by means of joining spatial fault configurations and value fault configurations into spatial-value configurations.

Definition 4.1.10 (Forcing Function)**.** The forcing function, denoted by φ, maps fault injection locations $l \in L$ to individual values $v \in V_u$. So,

$$\varphi : \; L \longrightarrow V_u, \; l \longmapsto v. \tag{4.3}$$

Since the unaffected value is included in the value configuration space V_u, fault injection locations can be configured individually to be affected by fault injection, and if affected, then a faulty value can be configured individually for each fault injection location. This allows to configure arbitrary single and multiple faults. The forcing function is defined in the spatial-value configuration space as follows.

Definition 4.1.11 (Spatial-Value Configuration Space)**.** The function space in which φ is defined, denoted by ${V_u}^L$, constitutes the spatial-value configuration space. It joins both the spatial configuration space and the value configuration space and includes all combinations of both spaces according to combinatorics. That is, there are $|V_u|^{|L|}$ distinct forcing functions φ defined in ${V_u}^L$. So,

$$\varphi \in {V_u}^L \tag{4.4}$$

Note, the forcing function φ that maps every $l \in L$ to $u \in V_u$, i.e. the case that all fault injection locations $l \in L$ remain unaffected, and therefore, no fault is injected, is denoted by φ_u, where

$$\forall \; l \in L \;:\; \varphi_u(l) = u. \tag{4.5}$$

Next, to determine the spatial property of a fault configuration, i.e. the spatial fault configuration L_a, from a known forcing function φ, the operator $L_a(\varphi)$ is defined.

$$L_a(\varphi) \; = \{\, l \mid \varphi(l) \neq u \,\}, \tag{4.6}$$

where $l \in L$ and $u \in V_u$ and $\varphi \in {V_u}^L$. The operator $L_a(\varphi)$ determines the set of affected fault injection locations L_a for which φ maps l to $v \neq u$. That is, $L_a = L_a(\varphi)$. Note, in case that $|V_u| = 2$, which is the case for bit-flip fault models, the number of distinct $L_a \in \mathcal{P}(L)$ matches exactly the number of distinct $\varphi \in {V_u}^L$. That is, if $|V_u| = 2$, then $|\mathcal{P}(L)| = \left|{V_u}^L\right|$.

Next, I describe the relation between spatial-value configuration space and the temporal granularity, for which I introduce a fault configuration function.

4.1.5 Total Configuration Space

The fault configuration function is defined in the total configuration space. Its purpose is to join the spatial configuration space and the value configuration space with temporal properties, which completes the meta fault configuration model. Important to note is that the meta fault configuration model allows to configure arbitrary permanent and transient faults at arbitrary fault injection times, which includes multiple fault injection times. This is further detailed in Section 4.3.

Definition 4.1.12 (Fault Configuration Function)**.** The fault configuration function, denoted by f, joins spatial, value and temporal components of the meta fault configuration model. For this purpose, the fault configuration function f maps time intervals $T_i \in T$ to individual forcing function $\varphi \in {V_u}^L$, where the forcing function φ maps fault injection locations $l \in L$ to individual values $v \in V_u$. So,

$$f: \; T \longrightarrow {V_u}^L, \; T_i \longmapsto \varphi. \tag{4.7}$$

Finally, the meta fault configuration model is completed with the definition for the total configuration space.

Definition 4.1.13 (Total Configuration Space)**.** The function space in which f is defined, denoted by $\left({V_u}^L\right)^T$, constitutes the total configuration space. So,

$$f \in \left({V_u}^L\right)^T. \tag{4.8}$$

Every fault configuration function f defined in $\left({V_u}^L\right)^T$ constitutes a distinct fault configuration, which describes a sequence of spatial-value configurations for consecutive time intervals.

The fault configuration function f that maps every $T_i \in T$ to $\varphi_u \in {V_u}^L$ represents the fault-free case and is denoted by f_u. In the fault-free case all fault injection locations $l \in L$ remain unaffected for all time intervals, and therefore, no fault is injected.

$$\forall\ T_i \in T\ :\ f_u(T_i) = \varphi_u. \tag{4.9}$$

Since the fault configuration function maps any time interval to arbitrary forcing functions, it allows to coherently define and handle all configuration possibilities. This includes arbitrary multiple fault injection times for which different or the same spatial and value configurations can be applied, which in turn includes arbitrary fault durations. This is further detailed in Section 4.4, where I discuss the fault injection complexity. In order to treat uniformly fault configuration functions that configure single or multiple affected time intervals, the temporal property of a fault configuration is defined as follows.

Definition 4.1.14 (Temporal Fault Configuration)**.** The set T_a denotes the temporal fault configuration, which models the temporal property of physical faults. T_a includes all affected time intervals and is a subset of T, which includes all time intervals. So

$$T_a = \{\, T_i \mid T_i \text{ is affected by faults, } T_i \in T \,\}, \tag{4.10}$$

where $T_a \subseteq T$.

Next, to determine the temporal fault configuration T_a of a known fault configuration function f, the operator $T_a\,(f)$ is defined.

$$T_a\,(f)\; = \{\, T_i \mid f(T_i) \neq \varphi_u \,\}, \tag{4.11}$$

where $0 \leq i < N_T$ and $T_i \in T$ and $\varphi_u \in {V_u}^L$ and $f \in \left({V_u}^L\right)^T$.

4.2 Fault Multiplicities

In this section I introduce terms commonly used in literature to describe the multiplicity of faults w.r.t. both the spatial and the temporal fault property. First, I discuss the spatial fault multiplicity, after which I discuss the temporal fault multiplicity.

Faults can be injected at single or multiple locations and at single or multiple times. Configuring multiple fault injection times is relevant from a practical point of view, e.g. for mimicking multiple event effects (MEE). Furthermore, in a security context, second order fault attacks using for example a single laser or multiple lasers to inject faults at two different times [TK10] motivate such configuration possibilities. In this case, multiple faults are injected physically in a time-displaced manner. Respective fault experiments are covered by the presented fault configuration model by means of configuring the composition of spatial and temporal properties of to be modeled faults.

4.2.1 Spatial Fault Multiplicity

The term spatial fault multiplicity (often referred to as fault multiplicity in literature), is commonly used in literature to clearly describe the number of affected fault injection locations.

Definition 4.2.1 (Spatial Fault Multiplicity)**.** The number of affected fault injection locations of a spatial fault configuration L_a is referred to as spatial fault multiplicity, denoted by m. The spatial fault multiplicity m is evaluated by the cardinality of the spatial fault configuration L_a. L_a can be determined with the operator $L_a\left(\varphi\right)$ (refer to Definition 4.1.6 and Equation 4.6). So,

$$m = |L_a| = \left|L_a\left(\varphi\right)\right|, \tag{4.12}$$

where $m \in \{0, ..., N_L\}$.

Note, since different time intervals may map to different forcing functions, the spatial fault multiplicity can be different for each time interval. This is relevant when considering transient faults, especially when multiple fault injection times are considered.

Definition 4.2.2 (No Fault)**.** If $m = 0$, then a fault-free (unaffected) case is described. In this case, the fault-free forcing function φ_u maps every given fault injection location $l \in L$ to the fault-free value $u \in V_u$, i.e. not a single fault injection location is affected, and therefore, no fault is injected.

Definition 4.2.3 (Single Fault)**.** If $m = 1$, then the respective fault is referred to as single fault since exactly one fault injection location is affected.

Definition 4.2.4 (Multiple Fault)**.** If $m > 1$, then the respective fault is referred to as multiple fault since multiple fault injection locations are simultaneously affected.

4.2.2 Temporal Fault Multiplicity

In order to clearly describe the number of affected time intervals of fault experiments, the term temporal fault multiplicity (also referred to as temporal multiplicity in literature) is commonly used in literature. That is, the temporal fault multiplicity describes a property of an entire fault experiment, which maps to a particular fault configuration in the context of the presented model.

Definition 4.2.5 (Temporal Fault Multiplicity)**.** The number of affected time intervals of a fault configuration is referred to as temporal fault multiplicity, denoted by k. The temporal fault multiplicity k is evaluated by the cardinality of the temporal fault configuration T_a, which includes all affected time intervals and can be determined with the operator $T_a\left(f\right)$ (refer to Definition 4.1.14 and Equation 4.11). So,

$$k = |T_a| = |T_a\left(f\right)|, \tag{4.13}$$

where $k \in \{0, ..., N_T\}$.

Definition 4.2.6 (Fault-free Configuration)**.** If $k = 0$, then a fault-free (unaffected) case is described. In the fault-free case, the fault configuration function f_u maps every given time interval T_i to the fault-free forcing function $\varphi_u \in {V_u}^L$. Hence, not a single fault injection location is affected at any time, and therefore, no fault is injected.

Definition 4.2.7 (Single Affected Time Interval)**.** If $k = 1$, then the respective fault configuration function f configures exactly one affected time interval, i.e. one time interval maps to a forcing function $\varphi \neq \varphi_u$.

Definition 4.2.8 (Multiple Affected Time Intervals)**.** If $k > 1$, then the respective fault configuration function f configures multiple affected time intervals, i.e. more than one time interval map to a forcing function $\varphi \neq \varphi_u$.

4.3 Practice-Oriented Interpretation

The presented meta fault configuration model defines fault injection for discrete time intervals. Whether or not a fault configuration actually results into errors depends on the circuit's input and its internal state, modeling of which is not part of the presented model. That is, the model does not describe the behavior of the system to which it can be applied, neither its fault-free nor its faulty behavior. This is intended to decouple modeling circuit behavior and modeling fault injection, which maximizes the flexibility in terms of applicability.

For the remainder of this work, a fault configuration function f and the function space $\left({V_u}^L\right)^T$ in which it is defined are interpreted as follows: Every defined time interval maps to a forcing function φ, which in turn maps all fault injection locations to an individual value for the respective time interval. This way, permanent and transient faults can be modeled, which includes faults that affect multiple, arbitrary locations for multiple, arbitrary fault injection times and arbitrary fault durations. Note that this allows to model physical faults that are injected using two or more independent sources, e.g. second-order attacks and even more sophisticated attacks. Moreover, the fault model type can be chosen arbitrarily from the defined value configuration space.

Next, I discuss the temporal fault properties fault injection time, fault release time and fault duration in the context of the presented meta fault configuration model. This closes the gap between the common understanding of what a fault is, as discussed in Section 2.2.2, and how these are described with the meta fault configuration model. Then, I introduce the term number of fault injection times, which I clearly distinguish from the temporal fault multiplicity. Finally, a description for a parametrized fault is given, which can be used to describe fault experiments with a single fault injection time.

4.3.1 Mapping Temporal Properties of Faults

The terms fault injection time and fault release time can be used to describe temporal properties of physical faults and modeled faults (refer to the discussion in Section 2.2.2). This makes sense as well in the context of the meta fault configuration model, however, only when considering a single time interval or multiple consecutive time intervals that map to the same $\varphi \neq \varphi_u$. Then, for all these consecutive time intervals the same spatial and value properties are considered, i.e. a single fault injection time is considered.

Definition 4.3.1 (Fault Injection Time)**.** A fault injection time of a fault configuration is denoted by t_{inj} and is the instance of time when the spatial or value property of a fault configuration changes such that the time interval T_i that includes t_{inj} maps to a $\varphi \neq \varphi_u$. Moreover, if a fault injection time is included in T_i, so $t_{\text{inj}} \in T_i$, then the respective time interval is hereinafter referred to as a fault injection interval and t_{inj} is the instance of time that defines the lower closed interval bound of the fault injection interval $T_i = [t_i, t_{i+1})$. So $t_{\text{inj}} = t_i$, where $0 \leq i < N_T$, i.e. $t_i < t_{N_T}$, and hence, $t_{\text{inj}} < t_{N_T}$. The instance of time t_i of a time interval T_i is a fault injection time if the time interval T_i in which t_i is included corresponds to either

- the very first time interval T_0 in case that T_0 maps to a $\varphi \neq \varphi_u$, so $f(T_0) \neq \varphi_u$, or
- a time interval T_i that maps to a $\varphi \neq \varphi_u$ and T_{i-1} maps to φ_u, so $f(T_i) \neq \varphi_u$ and $f(T_{i-1}) = \varphi_u$, or
- a time interval T_i that maps to a $\varphi \neq \varphi_u$ and T_{i-1} maps also to a $\varphi \neq \varphi_u$, however, the respective consecutive time intervals map to different $\varphi \neq \varphi_u$, so $f(T_i) \neq \varphi_u$ and $f(T_{i-1}) \neq \varphi_u$ and $f(T_i) \neq f(T_{i-1})$.

Note that the third case covers the second case. However, the second case is listed separately in order to highlight the difference between the transition from the fault-free configuration to a faulty configuration and the transition from one faulty configuration to another faulty configuration. A minimized description covering both the second and the third case is $f(T_i) \neq \varphi_u$ and $f(T_i) \neq f(T_{i-1})$, which is used hereinafter.

All fault injection intervals of a fault configuration are gathered in the set T_{inj}. So, $T_{\text{inj}} = \{\, T_i \mid t_{\text{inj}} \in T_i \,\}$, where $T_{\text{inj}} \subseteq T_a$. Next, in order to determine the set of fault injection intervals T_{inj} from a given fault configuration function f, the operator $T_{\text{inj}}(f)$ is defined.

$$\begin{aligned} T_{\text{inj}}(f) = &\{\, T_i \mid f(T_i) \neq \varphi_u \text{ and } f(T_i) \neq f(T_{i-1}) \,\} \\ &\cup \{\, T_0 \mid f(T_0) \neq \varphi_u \,\}, \end{aligned} \tag{4.14}$$

where $0 < i < N_T$ and $T_i \in T$ and $\varphi \in {V_u}^L$ and $f \in \left({V_u}^L\right)^T$.

Definition 4.3.2 (Fault Release Time)**.** A fault release time of a fault configuration is denoted by t_{release} and is the instance of time when the spatial and value property of a fault configuration is released. Moreover, if a fault release time is included in T_i, so $t_{\text{release}} \in T_i$, then the respective time interval is hereinafter referred to as a fault release interval and the fault release time is the instance of time that defines the upper open interval bound of the fault release interval $T_i = [t_i, t_{i+1})$ in which the fault is released. So $t_{\text{release}} = t_{i+1}$, where $t_{\text{inj}} < t_{\text{release}} \leq t_{N_T}$. The instance of time t_{i+1} of a time interval T_i is a fault release time if the time interval T_i in which t_{i+1} is included corresponds to either

- the very last time interval T_{N_T-1} in case that T_{N_T-1} maps to a $\varphi \neq \varphi_u$, so $f(T_{N_T-1}) \neq \varphi_u$, or
- a time interval T_i that maps to a $\varphi \neq \varphi_u$ and T_{i+1} maps to φ_u, so $f(T_i) \neq \varphi_u$ and $f(T_{i+1}) = \varphi_u$, or
- a time interval T_i that maps to a $\varphi \neq \varphi_u$ and T_{i+1} maps also to a $\varphi \neq \varphi_u$, however, the respective consecutive time intervals map to different φ , so $f(T_i) \neq \varphi_u$ and $f(T_{i+1}) \neq \varphi_u$ and $f(T_i) \neq f(T_{i+1})$.

Note, if two consecutive time intervals map to different spatial or different value properties, i.e. they map to different forcing functions φ, then the old spatial-value configuration is released while the new spatial-value configuration is applied. Hence, a specific instance of time can be both a fault release time and a fault injection time.

Next, the fault duration is introduced. So far, I intentionally omitted to consider the fault duration as temporal property of fault configurations. In the context of the presented meta fault configuration model, the fault duration implicitly follows from a sequence of the same spatial-value fault configuration for consecutive time intervals.

Definition 4.3.3 (Fault Duration)**.** The fault duration, denoted by d, of a fault configuration is defined by a fault injection time t_{inj} and the next fault release time t_{release} (refer to Definitions 4.3.1 and 4.3.2), where $d = t_{\text{release}} - t_{\text{inj}}$ and $d \in \mathbb{R}$ and $t_{\text{inj}} < t_{\text{release}}$. The fault duration interval, denoted by $T_d = [t_{\text{inj}}, t_{\text{release}})$, includes the respective affected consecutive time intervals T_i. All time intervals included in the fault duration interval map to the same φ, where $\varphi \neq \varphi_u$. That is,

$$\begin{gathered} \forall\, T_i \subseteq [t_{\text{inj}}, t_{\text{release}}) : T_d = \bigcup T_i \\ \text{and} \\ \forall\, T_{i+1}, T_i \subseteq T_d : \varphi = f(T_i) \text{ and } f(T_i) = f(T_{i+1}) \end{gathered} \tag{4.15}$$

where $T_i \in T$ and $\varphi \in {V_u}^L$ and $f(T_i) \in \left({V_u}^L\right)^T$.

In case of modeling physical faults with multiple fault injection times, the respective physical fault durations may overlap in time. This is modeled as composition of all fault properties. As a consequence, additional fault release times and fault injection times are created when physical fault durations start and end to overlap. Hence, several fault duration intervals T_d exists. Furthermore, if multiple physical sources affect the same locations, an application specific resolution function would be required to resolve this. Note that this is not relevant for the remaining content of this thesis, and therefore, I omit to introduce a description for multiple fault duration intervals and a resolution function.

As far as single fault injection times are concerned, a particular physical fault is mapped onto affected fault injection locations at the fault injection time t_{inj} until the fault release time t_{release}. Both together, t_{inj} and t_{release}, describe temporal properties of the modeled fault, from which also follows the fault duration d. Note that for single injection times a single fault duration interval T_d exists and it equals the set of affected time intervals T_a, so $T_d = T_a$ for single fault injection times.

Hereinafter, I consider the term fault duration only in case that a single fault injection time is considered. For this purpose, the Definition 4.3.3 is sufficient without introducing multiple fault duration intervals since properties of physical faults can be mapped directly to properties of the presented fault configuration model and vise verse. Note that the ability to model arbitrary multiple fault injection times is nevertheless covered by the total configuration space.

4.3.2 Number of Fault Injection Times

Changing spatial or value fault configurations during fault experiments is modeled by mapping the respective time intervals to a different forcing functions, which was introduced as fault injection time in Definition 4.3.1. The number of fault injection times describes how often the spatial or value fault configurations is changed during fault experiments. The number of fault injection times is also used later in Section 4.4 to derive equations for determining fault injection spaces and the respective fault injection complexity when considering a certain spatial fault multiplicity and a certain number of fault injection times. For this, the equations, developed in Section 4.4, are parametrized with the spatial fault multiplicity and with the number of fault injection times. As discussed in Section 4.3.1, when modeling time-displaced fault injection of physical faults, the number of modeled fault injection times is greater than the number of physical fault injection times in case that the physical fault injections overlap in time.

Definition 4.3.4 (Number of Fault Injection Times)**.** The number of fault injection times n is determined by the cardinality of the set of fault injection times T_{inj} which is determined by the operator $T_{\text{inj}}(f)$ (refer to Definition 4.3.1), where $t_{\text{inj}} \in T_{\text{inj}}$. So,

$$n = |T_{\text{inj}}| = |T_{\text{inj}}(f)|\,, \tag{4.16}$$

where $n \in \{0, ..., N_T\}$. Furthermore, $n \leq k$ since $T_{\text{inj}} \subseteq T_a$.

Definition 4.3.5 (No Fault Injection Time)**.** If $n = 0$, then a fault-free (unaffected) case is described. In the fault-free case, the fault configuration function f_u maps every given time interval T_i to the fault-free forcing function φ_u. Hence, not a single fault injection location is affected at any time, and therefore, no fault is injected.

Definition 4.3.6 (Single Fault Injection Time)**.** If $n = 1$, then the respective fault configuration f describes a single fault injection time. The modeled fault is originated either from a single source or from multiple sources that simultaneously causes the fault. Multiple consecutive time intervals may be affected (refer to Definition 4.2.8) in case that these map to the same forcing function φ (refer to Definition 4.3.3).

Definition 4.3.7 (Multiple Fault Injection Times)**.** If $n > 1$, then the respective fault configuration f describes multiple fault injection times. The modeled faults are originated from a single source or multiple sources in a time-displaced manner. This is modeled as a composition of these faults, where two different cases can be separated. Either the fault durations of to be modeled faults do not overlap in time or overlap in time. In case that consecutive time intervals are affected, at least two time intervals map to different spatial properties and/or different value properties, so $\varphi = f(T_i)$ and $f(T_i) \neq f(T_{i+1})$ (refer to Definition 4.3.1).

4.3.3 Permanent vs. Transient Faults

Permanent faults are described by fault configurations that map all time interval $T_i \in T$ to the same forcing function $\varphi \in {V_u}^L \setminus \varphi_u$. Hence, every affected fault injection location l maps to an individual value $v \neq u$ from the very first instance of time $t = t_0$ until the very last instance of time $t_{\text{release}} = t_{N_T}$. That is, all time intervals are affected, so $T_a = T$. From this also follows that the temporal fault multiplicity of permanent faults is $k = |T|$, whereas the number of fault injection times is $n = |T_{\text{inj}}| = 1$.

In contrast, a transient fault is described by fault configurations that map at least one $T_i \in T$ to the fault-free forcing function φ_u. In case that a composition of transient faults is modeled, at least one $T_i \in T$ exists which maps either to the fault-free forcing function φ_u or to a forcing function φ that is different to the one mapped to in the next time interval, i.e. $f(T_i) \neq f(T_{i+1})$ where $\varphi = f(T_i)$. Hence, the temporal fault multiplicity of transient faults is $k \leq |T|$, whereas the number of fault injection times is $n \geq 1$.

4.3.4 Parametrized Fault

Based on previous considerations, a family of fault configuration functions f_F is introduced for the purpose of describing fault configurations that consider solely single fault injection times, i.e. $n = 1$, with a variable fault duration.

First, I introduce a parametrized fault F as a tuple, which constitutes the short description of a particular fault configuration included in f_F.

Definition 4.3.8 (Parametrized Fault)**.** A parametrized fault, denoted by F, is a tuple composed of a particular forcing function $\varphi' \in {V_u}^L$, which maps the spatial property onto a value configuration space, and the temporal properties represented by the fault duration interval $T_d = [t_{\text{inj}}, t_{\text{release}})$.

$$F = (\varphi', T_d) \tag{4.17}$$

Note that all time intervals included in T_d map to the same φ'. Furthermore, all other time intervals not included in the defined fault duration interval T_d map to the fault-free forcing function φ_u, i.e. all $l \notin L_a$ map to the fault-free value $v = u$, so

$$\begin{gathered} \forall\, T_i \nsubseteq T_d : f(T_i) = \varphi_u \\ \text{and} \\ \forall\, T_i \subseteq T_d : f(T_i) \neq \varphi_u. \end{gathered} \tag{4.18}$$

The parametrized fault F is the corresponding description to the one used in literature to describe faults. In contrast to descriptions usually used in literature, the parametrized fault F can be used to describe permanent as well as transient faults uniformly.

Finally, the parametrized fault F is described in the context of the meta fault configuration model as a family of fault configuration functions f_F describing the subset of the total configuration space $\left(V_u{}^L\right)^T$ that includes all fault configurations with single fault injection times, i.e. $n = 1$.

Definition 4.3.9 (Fault Configuration Function for Single Fault Injection Time)**.** The fault configuration function for single fault injection times with configurable fault duration, denoted by f_F, where $f_F \in \left(V_u{}^L\right)^T$, defines a family of fault configuration functions that maps a fault duration interval $T_d = [t_{\text{inj}}, t_{\text{release}})$ to a particular forcing function $\varphi' \neq \varphi_u$, where $t_{\text{inj}}, t_{\text{release}} \in \mathbb{R}$ and $t_{\text{inj}} < t_{\text{release}}$ and $T_d = T_a$. All the other time intervals $T_i \nsubseteq T_d$, where $T_i \in T$, map to φ_u. So,

$$f_F\left(T_i\right) = \begin{cases} \varphi' \neq \varphi_u & \text{for } T_i \in T_d \\ \varphi_u & \text{otherwise} \end{cases}. \tag{4.19}$$

This way, an arbitrary spatial fault configuration for a single fault injection time t_{inj} with the fault duration $d = t_{\text{release}} - t_{\text{inj}}$ is configurable.

4.4 Fault Injection Complexity

In this section I discuss the complexity of fault injection campaigns along with the presented fault configuration model, separated into spatial, spatial-value and total fault injection complexity. Formal descriptions are presented that allow to evaluate the fault injection complexity when the sets L, V_u and T are given. One goal here is to create a comprehensive understanding about the presented meta fault configuration model's power.

The following discussion also aims on highlighting the complexity of fault modeling, here shown along with the presented configuration model. As also known from literature, the spatial fault injection complexity already results in a combinational explosion. I am going to show that the total fault injection complexity when considering value and temporal fault configurations in addition can be expressed by applying the binomial theorem three times. Note that fault injection is only one discipline that is involved for fault modeling; There are also others like modeling circuit behavior and test generation, which add further dimensions to it. This emphasizes the importance of robust fault selection strategies allowing to focus on relevant situations. But it also shows the importance of developing powerful and fast fault modeling tools, such as the methods I present in Chapters 6 to 8. These, on the one hand, require to provide configuration ability for relevant and, as appropriate, complex situations for security verification. On the other hand, fault modeling tools require to provide high performance.

The equations presented subsequently are derived from the presented meta fault configuration model and provide a powerful tool to determine the fault injection complexity. I am going to evaluate the fault injection complexity for practice-oriented subsets of the configuration space that follow from considering faults with a certain spatial fault multiplicity m (m-location fault) and a certain number of fault injection times n (n-injection times). This allows to evaluate for example the fault injection complexity when injecting all single faults or all multiple faults with a certain spatial fault multiplicity m for an arbitrary number of n fault injection times. Such complexity calculations are especially of importance to determine whether or not more complex fault injection campaigns are feasible in given verification time.

Important to note is that all configuration spaces of the presented meta fault configuration model include a fault-free case. When determining the complexity of fault injection campaigns, only cases where actually faults are injected are of interest. Therefore, to get the fault injection spaces, the fault-free cases need to be excluded from the corresponding configuration spaces.

4.4.1 Spatial Fault Injection Complexity

Next, the spatial fault multiplicity m is used to determine subsets of the spatial configuration space for m-location faults (e.g. all 2-location faults).

m-Location Faults The spatial configuration space $\mathcal{P}(L)$, defined in Definition 4.1.4, can be split into disjoint subsets that are expressed in terms of the spatial fault multiplicity m. Then, the m-th subset, denoted by $\mathcal{F}_{\text{spatial},m}$, where $\mathcal{F}_{\text{spatial},m} \subset \mathcal{P}(L)$, defines all spatial configuration possibilities where exactly m fault injection locations are affected. That is, $\mathcal{F}_{\text{spatial},m}$ includes all possible sets of affected fault injection locations L_a, where $L_a \in \mathcal{P}(L)$ and $|L_a| = m$, i.e. all m-combinations of $l \in L$ according to combinatorics. So,

$$\mathcal{F}_{\text{spatial},m} = \{\, L_a \mid |L_a| = m,\ L_a \in \mathcal{P} \,\}\,, \tag{4.20}$$

where $m \in \{\, 1, \ldots, N_L \,\}$. Note that $m \neq 0$. L_a can be determined with the operator $L_a\,(\varphi)\,$, so $L_a = L_a\,(\varphi)\,$, where $\varphi \in {V_u}^L$. The configuration possibilities covered by $\mathcal{F}_{\text{spatial},m}$ are then referred to as fault with spatial fault multiplicity m, or short m-location fault. This is a generic description for both single and multiple faults, which were defined in Definition 4.2.3 and Definition 4.2.4. This way, the fault-free case (no fault as per Definition 4.2.2) is excluded. The spatial fault injection space is introduced next.

Spatial Fault Injection Space The union of all $\mathcal{F}_{\text{spatial},m}$, where $m > 0$, evaluates the entire spatial fault injection space, which includes only configurations where fault locations are actually affected (exclusion of the empty set).

$$\mathcal{F}_{\text{spatial}} = \mathcal{P}(L) - \emptyset = \bigcup_{m=1}^{N_L} \mathcal{F}_{\text{spatial},m} \tag{4.21}$$

The first subset $\mathcal{F}_{\text{spatial},m=1}$ includes all spatial configuration possibilities where exactly one fault injection location is affected (single fault), and the second subset $\mathcal{F}_{\text{spatial},m=2}$ includes all spatial configuration possibilities where exactly two fault injection locations are affected (2-location fault), and so on.

The spatial fault injection complexity, equals the number of spatial fault configurations included in $\mathcal{F}_{\text{spatial}}$. It is evaluated by the cardinality of $\mathcal{F}_{\text{spatial}}$.

$$|\mathcal{F}_{\text{spatial}}| = |\mathcal{P}(L)| - 1 \tag{4.22}$$

There are two possibilities for each fault injection location; These are either affected or unaffected, and hence, $|\mathcal{F}_{\text{spatial}}|$ is evaluated by

$$|\mathcal{F}_{\text{spatial}}| = 2^{|L|} - 1. \tag{4.23}$$

Spatial Fault Injection Complexity for m-Location Faults From Equations 4.21 and 4.22 follows that the sum of all $|\mathcal{F}_{\text{spatial},m}|$ evaluates to the spatial fault injection complexity. The m-th summand, so the cardinality of the $\mathcal{F}_{\text{spatial},m}$, corresponds to the spatial fault injection complexity of all m-location faults.

$$|\mathcal{F}_{\text{spatial}}| = \left| \bigcup_{m=1}^{|L|} \mathcal{F}_{\text{spatial},m} \right| = \sum_{m=1}^{|L|} |\mathcal{F}_{\text{spatial},m}| \tag{4.24}$$

From combinatorics we know, when applying the binomial theorem, the number of q-elements subsets (q-combinations) of a p-element set is evaluated by the binomial coefficient $\binom{p}{q}$. In order to apply the binomial theorem, the term $2^{|L|}$ in Equation 4.23 is expressed as power of a binomial in the form $(x+y)^p$. Considering that each fault injection location l_j has one out of two possibilities, which are 'affected' and 'unaffected', we can assign $x = 1$ and $y = 1$, so

$$(x+y)^p = 2^{|L|} = (1+1)^{|L|}, \tag{4.25}$$

where $p = |L|$. By applying the binomial theorem with $p = |L|$ and $q = m$ we get

$$\begin{aligned} |\mathcal{F}_{\text{spatial}}| &= (1+1)^{|L|} - 1 \\ &= \sum_{m=1}^{|L|} \binom{|L|}{m} \cdot 1^{|L|+m} \cdot 1^m \\ &= \sum_{m=1}^{|L|} \binom{|L|}{m}. \end{aligned} \tag{4.26}$$

Note that $\binom{|L|}{0} = 1$ corresponds to the fault-free case, which is already excluded by evaluating the sum from $m = 1$ instead of from $m = 0$. Each summand, so the binomial coefficient $\binom{|L|}{m}$, corresponds to the spatial fault injection complexity when considering only faults with a specific spatial fault multiplicity m. That is, the spatial fault injection complexity of the m-th subset of $\mathcal{F}_{\text{spatial}}$ evaluates the spatial fault injection complexity of m-location faults. So,

$$|\mathcal{F}_{\text{spatial},m}| = \binom{|L|}{m}. \tag{4.27}$$

At this point I want to emphasize that the spatial fault injection complexity for moderate circuits with for example 1000 gates where each is considered to be a fault injection location, so $m = 1000$, yet alone cannot be handled exhaustively. That can be concluded from the fact that the spatial fault injection complexity is expressed as sum of binomial coefficients in Equation 4.26. However, it is still possible to select subsets for $m = 2$ for moderate circuits and $m = 3$ for important units of e.g. a processor. Furthermore, Equation 4.26 can be consulted for approximating the complexity for fault selection strategies based on layout information, where usually groups of cells are selected in which then multiple faults are configured.

Note that I have chosen to split $\mathcal{F}_{\text{spatial}}$ in subsets corresponding to a specific m, so $\mathcal{F}_{\text{spatial},m}$. However, arbitrary subsets that can be expressed by m can be chosen. For instance, when considering all 2-location and 3-location faults, so $\mathcal{F}_{\text{spatial},m=2} \cup \mathcal{F}_{\text{spatial},m=3}$, the spatial fault injection complexity can be determined by just adjusting the sum to iterate from $m = 2$ to 3.

Spatial Fault Injection Complexity for Single Faults The spatial fault injection complexity for single faults is then evaluated by the first summand according to Equation 4.26, so

$$\begin{aligned} \left|\mathcal{F}_{\text{spatial},m=1}\right| &= \sum_{m=1}^{1} \binom{|L|}{m} \\ &= |L| \, . \end{aligned} \tag{4.28}$$

Compared to multiple faults, the complexity of single faults is drastically reduced, and can also be handled with rather slow fault modeling techniques. It also constitutes the reason why fault injection campaigns usually start with single fault injection.

4.4.2 Spatial-Value Fault Injection Complexity

Next, the spatial-value fault injection space is discussed.

m-Location Faults Analogously to the considerations for the spatial configuration space presented in Section 4.4.1, the spatial-value configuration space $\mathcal{F}_{\text{spatial,value}}$ can be split into disjoint subsets that are expressed in terms of the spatial fault multiplicity m. The m-th subset of $\mathcal{F}_{\text{spatial,value}}$, denoted by $\mathcal{F}_{\text{spatial,value},m}$, includes all forcing functions $\varphi \in {V_u}^L$ that map exactly $m = \left|L_a\left(\varphi\right)\right|$ fault injection locations to a value $v \neq u$, i.e. all configuration possibilities where exactly m fault injection locations are affected. So,

$$\mathcal{F}_{\text{spatial,value},m} = \left\{ \varphi \,\middle|\, \left|L_a\left(\varphi\right)\right| = m, \ \varphi \in {V_u}^L \right\}. \tag{4.29}$$

Spatial-Value Fault Injection Space The union of all $\mathcal{F}_{\text{spatial,value},m}$, where $m > 0$, constitutes the entire spatial-value fault injection space, which covers only configurations where at least one fault location is affected. Therefore, the forcing function φ_u, which leaves all fault injection locations unaffected, has to be excluded. So,

$$\mathcal{F}_{\text{spatial,value}} = {V_u}^L \setminus \varphi_u = \bigcup_{m=1}^{N_L} \mathcal{F}_{\text{spatial,value},m} \, . \tag{4.30}$$

The spatial-value fault injection complexity is evaluated by the cardinality of the spatial-value fault injection space $\mathcal{F}_{\text{spatial,value}}$. It equals the number of forcing functions defined in the function space ${V_u}^L \setminus \varphi_u$. So,

$$|\mathcal{F}_{\text{spatial,value}}| = \left|{V_u}^L \setminus \varphi_u\right| = |V_u|^{|L|} - 1. \tag{4.31}$$

Spatial-Value Fault Injection Complexity for m-Location Faults Analogously to the spatial fault injection complexity, the binomial theorem is applied to determine the spatial-value fault injection complexity of m-location faults. According to Equations 4.31, the first term of $|\mathcal{F}_{\text{spatial,value}}|$ is evaluated by the $|L|$-th power of $|V_u|$. Since $V_u = u \cup V$, and therefore, $|V_u| = (1 + |V|)$, this can be expressed as power of a binomial in the form $(x + y)^p$. Let $x = 1$, $y = |V|$, $p = |L|$ and $q = m$, and by applying the binomial theorem we get

$$\begin{aligned} |V_u|^{|L|} &= (1 + |V|)^{|L|} \\ &= \sum_{m=0}^{|L|} \binom{|L|}{m} \cdot 1^{|L|+m} \cdot |V|^m. \end{aligned} \tag{4.32}$$

To get the spatial-value fault injection complexity, the forcing function φ_u that leaves all fault injection locations unaffected is excluded from ${V_u}^L$ according to Equation 4.31. Therefore, one is subtracted from $|V_u|^{|L|}$, which corresponds to excluding the case $m = 0$ from the sum, so

$$\begin{aligned} |\mathcal{F}_{\text{spatial,value}}| &= |V_u|^{|L|} - 1 \\ &= \sum_{m=1}^{|L|} \binom{|L|}{m} \cdot (|V_u| - 1)^m. \end{aligned} \tag{4.33}$$

The summands of the sum in Equation 4.33 consists of two factors. The first factor $\binom{|L|}{m}$ was already explained with Equation 4.26 and constitutes the spatial fault injection complexity for m-location faults. The second factor constitutes the contribution of the value configuration space to the spatial-value fault injection complexity. Each summand corresponds to the spatial-value fault injection complexity of the m-th subset of $\mathcal{F}_{\text{spatial,value}}$, which is denoted by $|\mathcal{F}_{\text{spatial,value},m}|$ and is evaluated by

$$|\mathcal{F}_{\text{spatial,value},m}| = \binom{|L|}{m} \cdot (|V_u| - 1)^m. \tag{4.34}$$

So, Equation 4.34 evaluates all possibilities to choose m out of $|L|$ fault injection locations to be affected, and furthermore, it considers all possible value assignments $v \neq u$ for affected locations. For instance, Equation 4.34 can be consulted to evaluate the fault injection complexity of all possible 2-location stuck-at faults, when plugging in $m = 2$ and $|V_u| - 1 = 2$.

Note, for the bit-flip fault model the contribution of the value granularity is eliminated since V_u includes only two elements, namely bit-flip and unaffected. For the stuck-at fault model, however, this factor is 2^m since V_u includes three elements, namely stuck-at-1, stuck-at-0 and unaffected. At this point, the contribution of the temporal properties is not yet considered. However, from this relation it can already be concluded that when modeling transient faults for the shortest possible duration of one time interval, the bit-flip fault model is considerably more efficient than the temporary stuck-at fault model.

Spatial-Value Fault Injection Complexity for Single Faults The spatial-value fault injection complexity for single faults is evaluated by the first summand according to Equation 4.33, so

$$\begin{aligned} |\mathcal{F}_{\text{spatial,value},m=1}| &= \sum_{m=1}^{1} \binom{|L|}{m} \cdot |V| \\ &= |L| \cdot (|V_u| - 1). \end{aligned} \tag{4.35}$$

Analogously to the discussion of m-location faults, the contribution of the value granularity is eliminated for bit-flip faults, whereas it is effective if V_u includes two or more elements, e.g. for stuck-at faults. However, for single faults it's contribution is not expressed as a power of a binomial anymore and therefore the Spatial-Value Fault Injection Complexity increases linearly with the value granularity. Note, when considering single faults, this allows to increase the value granularity of, e.g., fault models for analog circuits.

4.4.3 Total Fault Injection Complexity

Next, the number of fault injection times n is used to determine subsets of the total configuration space that include all fault configurations to configure n fault injection times during fault experiments.

n-Injection Times The total configuration space $\left({V_u}^L\right)^T$, defined in Equation 4.8, can be split into subsets described by the number of fault injection times n. Then, the n-th subset, denoted by $\mathcal{F}_{\text{total},n}$, defines all configuration possibilities where exactly n fault injection times are considered, where $n = |T_{\text{inj}}(f)|$ and $f \in \left({V_u}^L\right)^T$. That is, $\mathcal{F}_{\text{total},n}$ includes all configuration possibilities that are n-combinations of $T_i \in T$ according to combinatorics. So,

$$\mathcal{F}_{\text{total},n} = \left\{ f \,\middle|\, |T_{\text{inj}}(f)| = n,\ f \in \left(V_u{}^L\right)^T \right\}, \tag{4.36}$$

where $n \in \{\, 1, \ldots, N_T \,\}$. Configuration possibilities included in $\mathcal{F}_{\text{total},n}$ are then referred to as fault configuration with a number of fault injection times n, or short fault configurations for n-injection times. Note that $n \neq 0$. This way, the fault-free configuration f_u (refer to Equation 4.9) is excluded.

Next, the total fault injection space is discussed, which includes all total configurations that actually represent fault injection.

Total Fault Injection Space The configuration function $f \in \left(V_u{}^L\right)^T$ maps every time interval T_i to an individual, arbitrary forcing function $\varphi \in V_u{}^L$. Excluding the fault-free configuration f_{φ_u}, so the fault configuration function that maps all time intervals to φ_u as defined earlier in Equation 4.9, gives us the total fault injection space $\mathcal{F}_{\text{total}}$. This is equal to the union of all n-th subsets $\mathcal{F}_{\text{total},n}$, where $n \in \{\, 1, \ldots, N_T \,\}$. So,

$$\mathcal{F}_{\text{total}} = \left(V_u{}^L\right)^T \setminus f_{\varphi_u} \quad = \bigcup_{n=1}^{|T|} \mathcal{F}_{\text{total},n}. \tag{4.37}$$

$\mathcal{F}_{\text{total}}$ includes all configuration possibilities w.r.t. spatial, value and temporal properties where for at least one time interval a faulty value is mapped to at least one location. That is, for at least one time interval $f(T_i) \neq \varphi_u$. So,

$$\exists\, T_i \in T \ :\ f(T_i) \neq \varphi_u. \tag{4.38}$$

The number of fault configuration functions defined in $\mathcal{F}_{\text{total}}$ evaluates the total fault injection complexity of a fault model and corresponds to the number of fault experiments desired to be performed during fault injection campaigns.

$$\begin{aligned} |\mathcal{F}_{\text{total}}| &= \left| \left(V_u{}^L\right)^T \setminus f_{\varphi_u} \right| \\ &= \left(|V_u|^{|L|} \right)^{|T|} - 1 \end{aligned} \tag{4.39}$$

Now, Equation 4.39 demonstrates that it is impossible already for very small circuits to cover the total fault injection space exhaustively. Let $|V_u| = 2$, which is the case for the bit-flip fault model, $|L| = 100$, which means that the considered circuit has 100 fault injection locations and $|T| = 100$ representing the duration of a fault experiment. To cover all possible fault configurations for this very small circuit, already $|\mathcal{F}_{\text{total}}| = 1,995 \cdot 10^{3010}$

distinct fault configurations can be considered for fault injection campaigns, which motivates using fault selection strategies to select relevant subspaces.

Subsequently, I am going to split the total fault injection complexity with help of both the spatial fault multiplicity and the number of fault injection times into practice-oriented fractions relevant for fault selection based on for example layout information. The goal is to formulate m and n into a single equation, which allows determining the total fault injection complexity when considering m-location faults that are injected at n-injection times, which constitutes the m, n-th subset of $\mathcal{F}_{\text{total}}$.

First, I represent $\mathcal{F}_{\text{total}}$ in a form that depends on the spatial fault multiplicity m analogously to previous considerations for $\mathcal{F}_{\text{spatial}}$ and $\mathcal{F}_{\text{spatial,value}}$. This allows to determine the total fault injection complexity when considering m-location faults according to Equation 4.20. Then, $\mathcal{F}_{\text{total}}$ is represented in a form that depends on the number of fault injection times n. This allows to determine the total fault injection complexity when considering n-injection times according to Equation 4.36. Finally, I represent $\mathcal{F}_{\text{total}}$ in a form that depends on both parameters m and n. Based on these three equations, three corresponding equations are derived, allowing to parametrize $|\mathcal{F}_{\text{total}}|$ either with a particular m, a particular n or a combination of both.

First, I plug in $|\mathcal{F}_{\text{spatial,value}}|$ from Equation 4.33 into Equation 4.39 to get $|\mathcal{F}_{\text{total}}|$ dependent on the spatial fault multiplicity m. So,

$$\begin{aligned}
|\mathcal{F}_{\text{total}}| &= \left(|V_u|^{|L|}\right)^{|T|} - 1 \\
&= \left(|V_u|^{|L|} - 1 + 1\right)^{|T|} - 1 \\
&= (1 + |\mathcal{F}_{\text{spatial,value}}|)^{|T|} - 1 \\
&= \left(1 + \sum_{m=1}^{|L|} \binom{|L|}{m} \cdot (|V_u| - 1)^m \right)^{|T|} - 1.
\end{aligned} \tag{4.40}$$

Then, second, the binomial theorem is applied to Equation 4.39 to get $|\mathcal{F}_{\text{total}}|$ dependent on the number of fault injection times n. So,

$$\begin{aligned}
|\mathcal{F}_{\text{total}}| &= (-1) + \sum_{n=0}^{|T|} \binom{|T|}{n} \cdot |\mathcal{F}_{\text{spatial,value}}|^n \\
&= \sum_{n=1}^{|T|} \binom{|T|}{n} \cdot \left(|V_u|^{|L|} - 1\right)^n.
\end{aligned} \tag{4.41}$$

Finally, I plug in $|\mathcal{F}_{\text{spatial,value}}|$ from Equation 4.33 into Equation 4.41

to get $|\mathcal{F}_{\text{total}}|$ dependent on both the spatial fault multiplicity m and the number of fault injection times n.

$$\begin{aligned}|\mathcal{F}_{\text{total}}| &= (-1) + \sum_{n=0}^{|T|} \binom{|T|}{n} \cdot |\mathcal{F}_{\text{spatial,value}}|^n \\ &= \sum_{n=1}^{|T|} \binom{|T|}{n} \cdot \left(\sum_{m=1}^{|L|} \binom{|L|}{m} \cdot (|V_u| - 1)^m \right)^n .\end{aligned} \tag{4.42}$$

In Equation 4.40 to Equation 4.42, the total fault injection complexity now depends on the spatial fault multiplicity m and the number of fault injection times n. Next, three corresponding equations are derived that allow to parametrize the considered subset of $\mathcal{F}_{\text{total}}$ with m or n or both m and n, respectively. These parametrized equations are discussed and used to determine the complexity of specific subsets of the fault injection space that are relevant for following implementations of fault injection campaigns and the result chapter (Chapter 9).

Since I am going to choose the m-th, n-th and m,n-th subsets of $\mathcal{F}_{\text{total}}$, the sums iterating m and n in Equation 4.40 to 4.42 are collapsed to a single summand. So, we get

$$|\mathcal{F}_{\text{total},m}| = \left(1 + \binom{|L|}{m} \cdot (|V_u| - 1)^m \right)^{|T|} - 1, \tag{4.43}$$

$$|\mathcal{F}_{\text{total},n}| = \binom{|T|}{n} \cdot \left(|V_u|^{|L|} - 1\right)^n, \tag{4.44}$$

and

$$|\mathcal{F}_{\text{total},m,n}| = \binom{|T|}{n} \cdot \left(\binom{|L|}{m} \cdot (|V_u| - 1)^m \right)^n . \tag{4.45}$$

Now, $|\mathcal{F}_{\text{total}}|$ is parametrized with either m or n or with both parameters m and n. Based on these parametrized equations, the complexity of performing fault injection campaigns for practice-oriented subsets of $\mathcal{F}_{\text{total}}$ can be determined.

Subsequently, I am going to discuss the complexity when considering particular fault multiplicities (m-location and single faults) and particular number of fault injection timess (n-injection times and single fault injection time) with a fixed fault duration of $d = 1$ as well as configurable fault durations.

Total Fault Injection Complexity for m-Location Faults Evaluating $|\mathcal{F}_{\text{total},m}|$ for a specific m according to Equation 4.43 determines the fault injection complexity when considering all configuration possibilities that map all time intervals to all possible m-location faults. That is, for every possible time interval there are $1 + \binom{|L|}{m} \cdot (|V_u| - 1)^m$ possibilities considered. For example, evaluating $|\mathcal{F}_{\text{total},m}|$ for a particular $m = 2$ determines the fault injection complexity of the subset of the total fault injection space that includes all possible 2-location faults. Note, the left summand 1+ in the outer parentheses in Equation 4.43 corresponds to the contribution of fault-free time intervals, i.e. respective time intervals map to $\varphi_u \in {V_u}^L$. The right summand -1 corresponds to excluding the fault-free configuration f_u.

Total Fault Injection Complexity for Single Faults The total fault injection complexity for single faults, denoted by $|\mathcal{F}_{\text{total},m=1}|$, is then evaluated by plugging in $m = 1$ into Equation 4.43. So,

$$\begin{aligned} |\mathcal{F}_{\text{total},m=1}| &= \left(1 + \binom{|L|}{1} \cdot (|V_u| - 1)^1\right)^{|T|} - 1 \\ &= (1 + |L| \cdot (|V_u| - 1))^{|T|} - 1. \end{aligned} \tag{4.46}$$

Total Fault Injection Complexity for n-Injection Times Evaluating Equation 4.44 for a specific n determines the fault injection complexity for considering exactly n fault injection times. For example, plugging in $n = 2$ in Equation 4.44 allows to determine the total fault injection complexity of modeling all possible second order fault attacks, i.e. two time-displaced fault injections.

Total Fault Injection Complexity for Single Fault Injection Times To evaluate the total fault injection complexity for single fault injection times, denoted by $|\mathcal{F}_{\text{total},n=1}|$, $n = 1$ is plugged into Equation 4.44. This gives us

$$|\mathcal{F}_{\text{total},n=1}| = |T| \cdot \Big(|V_u|^{|L|} - 1\Big). \tag{4.47}$$

Note that the fault duration interval T_d, and therefore, the fault duration d is not specified and also its variation possibilities are not considered in the complexity calculations. Variable fault durations for single fault injection times are discussed later in this section.

Total Fault Injection Complexity for m-Location Faults and n-Injection Times So far, I discussed the fault injection complexity when $|\mathcal{F}_{\text{total}}|$ is parametrized by either m or n. However, when choosing a specific number of fault injection times n, only the temporal configuration possibilities are constraint, but the fault injection complexity is still determined for the entire spatial-value configuration space. Further, when choosing a specific m, a subset of the spatial-value configuration space is considered, but all possible temporal configurations are still considered for the fault injection complexity. As a consequence, most fault configurations contributing to the fault injection complexity are not relevant from a practical-oriented point of view and, hence, results of corresponding fault injection campaigns would be too pessimistic. For instance, due to the spot size of a laser, a laser fault attack may affect only a view cells at once and not all cells of the a circuit. In order to consider this, $|\mathcal{F}_{\text{total}}|$ needs to be parametrized with both m and n according to Equation 4.45. This way, we can determine the fault injection complexity for the case that n arbitrary fault injection times are chosen at which m arbitrary locations are affected. This constitutes the most powerful description and provides a tool to determine the complexity of fault injection campaigns that mimic complex situations such as second-order laser attacks.

Example 4.4.1 (Second Order Fault Attack)**.** Assume a laser is used to conduct a second order fault attack, for which two independent time-displaced shots are required. Furthermore, matching the spot size onto the layout, one can approximate how many cells may be affected at most. We now assume five cells are affected at most. Then, by plugging in $m = 5$ and $n = 2$, we can determine how many possibilities we have to consider for fault injection campaigns. Considering that a shot may also affect only subsets of five cells, we could also consider $m = 4$, $m = 3$, $m = 2$ and $m = 1$ and sum the respective complexities. Note that in this example it is assumed that both laser shots would affect the same amount of cells.

It is also possible to consider all possible combinations, where the first shot may affect a different amount of cells than the second shot. Instead of parametrizing $\mathcal{F}_{\text{total}}$ based on a particular m and n, another subset of $\mathcal{F}_{\text{total}}$ that can be described by these parameters can be chosen as well. For example, a range of fault multiplicities can be considered. Note that considering a range of multiplicities results in a corresponding sum of binomial coefficients again. This is especially important to note for the spatial fault multiplicity m since the corresponding sum is part of a binomial to the power of $|T|$ (power of n for the parametrized form).

Assume that a laser shot affects two to three cells and the laser shoots at two distinct times. Now, we chose m in the range $2, \ldots, 3$ and $n = 2$, where $\left|\mathcal{F}_{\text{total},m\in\{2,3\},n}\right|$ denotes the corresponding fault injection space. Then we get

$$\left|\mathcal{F}_{\text{total},m\in\{2,3\},n=2}\right| = \binom{|T|}{2} \cdot \left(\sum_{m=2}^{3} \binom{|L|}{m} \cdot (|V_u| - 1)^m \right)^2 . \tag{4.48}$$

Let us further assume that V_u includes two elements, one value representing the faulty and one value representing the unaffected case (this applies e.g. for bit-flip faults). Then we get $\left|\mathcal{F}_{\text{total},m\in\{2,3\},n=2}\right| = \binom{|T|}{2} \cdot \left(\sum_{m=2}^{3} \binom{|L|}{m}\right)^2$. Evaluating the sum gives us a binomial of the form $(a+b)^2 = a^2 + 2ab + b^2$, which means that either both shots affect two locations (a^2), both shots affect three locations (b^2) or one shot affects two locations whereas the other shot affects 3 locations or vise verse ($2ab$).

Total Fault Injection Complexity for Single Faults and Single Fault Injection Time The total fault injection complexity for single faults and single fault injection times, denoted by $|\mathcal{F}_{\text{total},m=1,n=1}|$, is evaluated by plugging in $m = 1$ and $n = 1$ into Equation 4.45, resulting in

$$|\mathcal{F}_{\text{total},m=1,n=1}| = |T| \cdot |L| \cdot (|V_u| - 1). \tag{4.49}$$

Note that the fault duration interval T_d and the fault duration d are still not considered, and therefore, also its variation possibilities are not considered in the complexity calculations. Configurable fault durations for single fault injection times are discussed next.

Total Fault Injection Complexity for Single Fault Injection Times and Varying Fault Durations The fault duration interval T_d can last from the fault injection time t_{inj} up to the very last discrete time t_{N_T}, i.e. $T_d \subseteq [t_{\text{inj}}, t_{N_T})$ and $d \leq (t_{N_T} - t_{\text{inj}})$. Now, to get the total fault injection complexity when considering all configuration possibilities for fault durations, $\binom{|T|}{2}$ valid fault durations need to be considered. According to combinatorics, this follows from choosing all possible two-combinations of discrete times, namely t_{inj} and t_{release}, from all possible times. To consider single fault injection times and all possible configurations defined in the spatial-value fault injection complexity as well as all possible fault durations for the complexity calculations, in Equation 4.47 $N_T = |T|$ possible fault injection times are replaced by $\binom{|T|}{2}$ valid fault durations. So,

$$|\mathcal{F}_{\text{total},n=1,\forall d}| = \binom{|T|}{2} \cdot \left(|V_u|^{|L|} - 1\right), \tag{4.50}$$

where $|\mathcal{F}_{\text{total},n=1,\forall d}|$ indicates that all possible fault durations are considered. The same consideration can be applied to all Equations that consider single fault injection times. Applying it to, e.g., Equation 4.49, which describes single faults and single fault injection times, results in $|\mathcal{F}_{\text{total},m=1,n=1,\forall d}|$.

$$|\mathcal{F}_{\text{total},m=1,n=1,\forall d}| = \binom{|T|}{2} \cdot |L| \cdot (|V_u| - 1) \tag{4.51}$$

4.5 Summary

A meta fault configuration model was developed, which constitutes a superset of fault models known from literature. For this purpose, its spatial configuration space, value configuration space and temporal properties as well as its granularity were defined independent of specific abstraction levels. The relation between the spatial and value configuration space have been described as forcing function, which maps fault injection locations individually to values defined in the value configuration space, resulting in a spatial-value configuration space. Finally, the relation between spatial-value configuration space and temporal properties was described as fault configuration function, which maps time intervals to arbitrary forcing functions. This way, a meta model was created that has the flexibility to be further specified for specific abstraction levels and fault model types, while it covers any fault configuration relevant to mimic physical fault behavior and especially arbitrary physical fault attacks.

Moreover, commonly used terms such as single and multiple faults, transient and permanent faults, the spatial and temporal fault multiplicity, fault injection spaces, the temporal properties fault injection and fault release time as well as multiple fault injection times and fault durations were formulated in the context of this model and their practical relevance was discussed.

The fault injection complexity was determined for the spatial, spatial-value and total fault injection spaces. These configuration spaces have been split, parametrized by both the spatial fault multiplicity and the number of fault injection times, into subsets relevant from a practical point of view. This way, fault injection spaces and the corresponding fault injection complexity for sophisticated fault injection campaigns that for example mimic second order fault attacks can be determined. Hence, besides a very flexible meta fault configuration model, powerful and practice-oriented tools have been presented.

Chapter 5

State-of-the-Art of FPGA-based Fault Emulation

In this chapter I detail the concepts of FPGA-based fault emulation along with the related work. The focus lies on creating an overview and reviewing existing implementations for the different tasks involved in fault emulation. I also discuss possibilities to implement components of fault emulation environments either in software or hardware, where I focus on respective advantages and disadvantages. In the process, I discuss which design decisions I have made to improve existing work with respect to configurability, performance and applicability to security verification.

I start with a simplified illustration in Section 5.1 to give an overview. In Section 5.2, I discuss different fault injection methodologies. Section 5.3 to Section 5.5 detail fault generation, test generation and fault classification for fault emulation.

After reviewing the existing work in this chapter, in the next Chapter 6 I present concepts and implementation details of the FPGA-based fault emulation environment that I propose for security verification. Fault injection in combinational logic is improved by extending fault emulation by a software-based pre-processor in Chapter 7. Performance optimizations are introduced in Chapter 8, which close the gap between performance and configurability. Furthermore, these enable to configure arbitrary multiple fault injection times.

5.1 Concept

For FPGA-based fault emulation the Circuit Under Verification (CUV) is synthesized onto an FPGA. Usually, fault emulators are embedded into a

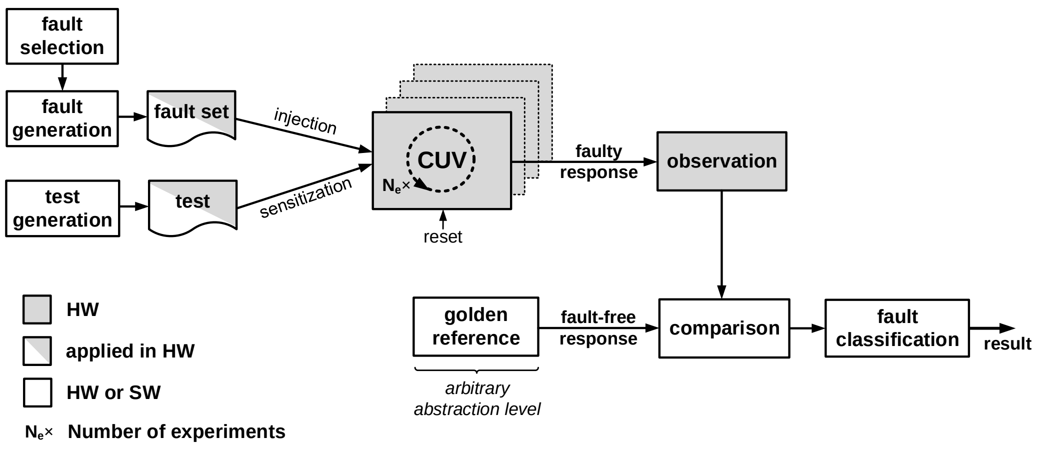


Figure 5.1: Concept of FPGA-based fault emulation.

fault emulation environment composed of additional software components. These control the communication with the fault emulator and take care of pre- and post-processing steps, such as FPGA synthesis, adding fault injection capability to the CUV, test and fault generation as well as upload, and result download. To sensitize the CUV during fault injection campaigns, the CUV processes input stimuli, which are either a structural test or a functional test software (depends on the CUV's architecture). During test execution, faults are injected into the CUV based on a fault model implementation in hardware, which is the key task of fault emulation. In the process, the CUV's response is observed and compared to a fault-free reference. Based on this comparison, fault classification is performed, whose result is reviewed by a verification engineer.

The concept of FPGA-based fault emulation is depicted in Figure 5.1. Characterizing for fault emulation is that the CUV is implemented in an FPGA, indicated by the gray colored blocks. Fault injection and test execution is also implemented in hardware on the FPGA, and therefore, it is required that at least one fault configuration as well as one test pattern is stored and then applied in hardware, indicated by the half-gray colored blocks in Figure 5.1. However, various tasks can be implemented either in hardware or software (white colored blocks). These include fault selection, fault generation, test generation, generation of a fault-free references, response observation, comparison versus a fault-free reference and fault classification.

Since the communication interface is the performance limiting factor of fault emulation, communication between software and hardware components needs to be minimized [LGPE05]. The more is implemented in software, the more data is frequently exchanged between software and hardware components. This causes idle times in hardware execution during which the fault emulation is waiting for new fault configurations. In the worst case, the hardware is waiting for new data in between all consecutive fault experiments. Hence, it is preferred to integrate as much as possible into hardware to minimize the communication overhead. However, implementing parts in software provides more flexibility, which might be required for example when considering layout information for fault generation. In summary, implementing parts in software is more flexible but slower, whereas corresponding hardware implementations are faster but less flexible.

The subsequent sections give detailed descriptions for the basic components depicted in Figure 5.1 and related tasks of fault emulation. The focus lies on discussing advantages and drawbacks of software and hardware implementations.

5.2 Fault Injection Methodologies

Fault injection capability is realized by altering the hardware of the CUV, which can be realized at three different hardware abstraction levels: RTL, gate level or FPGA configuration bitstream level.

Three corresponding techniques for fault injection are reported in the literature, based on which fault emulation techniques are roughly divided:

- partial FPGA-reconfiguration (configuration bitstream level)
- mutant- and saboteur-based HDL modification (RT level)
- instrumented circuit technique (gate level)

These are subsequently detailed along with the related work.

5.2.1 Partial FPGA-reconfiguration

Exploiting FPGA-reconfiguration for fault grading was the initial fault emulation approach. It is suitable for fault injection in combinational and sequential cells. Cheng et al. [CHD95, CHD99] proposed compile-time FPGA reconfiguration to model permanent faults based on the stuck-at fault model. Antoni et al. [ALF00] proposed local (also known as partial or dynamic) real-time FPGA reconfiguration in order to also enable modeling transient faults. Basically, this methodology implements a read-modify-write scheme for the frame of the FPGA configuration RAM in which the fault is injected. In case of transient faults, the hardware execution has to be interrupted in order to alter the FPGA configuration, which causes a time overhead, and therefore, results in performance loss.

This methodology is considered to be slow and less flexible compared to the other two approaches, especially when multiple faults are considered. Since the other fault injection methodologies better suit my goals with respect to applicability for security verification and performance, I do not consider this approach in this thesis.

5.2.2 Mutant-based and Saboteur-based HDL Modification

Mutants and saboteurs were originated in the RTL fault simulation domains [GC91, JAR+94]. For fault emulation these techniques are applied by HDL modifications.

Mutants replace the original component RTL description by a description that is capable of injecting faults. In contrast, saboteurs are components that are capable of altering signal behavior and extend an RTL design without replacing the original description [JAR+94]. Leveugle [Lev00] utilized mutant generation for an FPGA-based emulation environment. Baraza et al. [BGGG05] present an emulation environment capable of placing saboteurs and generating mutants. Baraza et al. also propose an automatic saboteur placement and mutant generation in order to gain a better performance, which was implemented by e.g. Grinschgl et al. [GKS+11a, GKS+11b] for saboteurs.

The advantage of mutant- and saboteur-based emulation environments is enabling early dependability analysis by using well known HDL, resulting in high flexibility with respect to implementing different fault models. Since HDL modifications are done pre-synthesis, this approach is only suitable for fault injection in sequential cells and signals described at RTL. Hence, this approach does not allow to model arbitrary faults in combinational cells at gate level, which is necessary for exhaustive fault injection in combinational logic. Further drawbacks are the required HW-overhead and synthesis time overhead [GKS+11a]. Due to the high HW-overhead, fault injection capability cannot be added to every possible location at once. Therefore, already multiple synthesis runs would be necessary to provide a full coverage of single faults. This does not only limit the overall performance drastically, it also renders the capability to select arbitrary multiple faults impossible. Therefore, these techniques are unattractive for fault selection strategies based on e.g. layout information, for which the selection of arbitrary multiple faults is mandatory. Since I am aiming for a fast and complete solution for modeling fault attacks in gate level netlists, I do not consider this approach.

5.2.3 Circuit Instrumentation Technique

Contrary to partial FPGA-reconfiguration and mutant- and saboteur-based HDL modifications, the instrumented circuit technique only requires a single time-consuming synthesis run and FPGA-configuration to prepare the CUV for fault injection. Furthermore, the instrumented circuit technique does not require HDL or library modifications, is applicable to gate level netlists and is fast. Therefore, I consider this technique in this thesis.

Circuit instrumentation focuses on adding fault injection capability for flip flops (FFs) or combinational cells within a gate level description. For this purpose, FFs or combinational cells are extended by additional logic in order to support fault injection based on fault models. This is referred to as circuit instrumentation. At runtime, either a single gate (single fault) or a set of instrumented gates (multiple fault) is selected to be faulty during a particular fault experiment. All the other gates keep their fault-free functionality.

Civera et al. [CMR$^+$01, CMR$^+$02] propose a register implemented as scan-chain to select faulty gates at runtime, the so called fault mask register. When the fault injection time is reached, all faults selected in the fault mask register are enabled for one clock cycle. For each fault experiment the entire content of the fault mask register is uploaded over the communication interface. The performance of this implementation depends on the number of instrumented gates since larger CUVs require more data for updating the fault mask register.

López-Ongil et al. [LGPE05, LGPE07] present two new techniques: the state-scan technique and the time-multiplexed technique. These techniques connect all FFs of the CUV to a scan-chain. The scan-chain is then loaded with a faulty state of the CUV, representing the fault injection.

Vanhauwaert et al. [VLR06] propose to load the fault mask register in parallel instead of using a scan chain approach. This reduces the time spent for preliminary configuration phases of fault experiments. Furthermore, Vanhauwaert et al. propose to represent the value of the fault mask register in binary code, instead of using an individual bit for each fault injection location in order to save sequential logic. However, a single binary coded fault mask register only allows to address single faults. Vanhauwaert et al. mention that introducing several binary coded fault mask registers, e.g. two fault mask registers for two bit faults, would fix this issue.

Janning et al. [JHSS11] propose to set an individual bit within the fault mask register for single fault selection. The location within the fault mask register is addressed by its corresponding index. Instead of uploading the entire fault mask register content, only the corresponding index of the fault injection location needs to be uploaded over the communication interface. This reduces the data upload on the communication interface for single faults. Janning et al. update the entire content of the fault mask register for multiple fault selection.

Earlier in 2011 [NR11], I proposed a more generalized concept that allows to also address arbitrary multiple locations for multiple fault injection. A refined and even more generalized approach and its implementation is presented in Section 6.3. It allows to implement different fault models for fault injection in sequential as well as combinational logic in a generic way. The subsequent sections discuss methods related to fault generation, test generation, response observation and fault classification.

5.3 Fault Generation and Upload

As discussed in Section 3.2, in order to use limited verification times efficiently, fault configurations are generated based on fault selection strategies. As depicted in Figure 5.1, fault generation can be realized either in software or hardware.

For fault injection of single or randomly selected multiple faults, it is possible to generate fault configurations automatically in hardware on the FPGA. This is also known as autonomous fault emulation, which was introduced by López-Ongil et al. [LGPE05, LGPE07]. However, autonomous fault emulation does not suit more complex fault injection campaigns, e.g. based on layout or structural information, which require capability to configure arbitrary multiple faults.

Fault emulation environments supporting arbitrary multiple fault injection therefore generate fault lists (also known as fault dictionaries) in software and upload these onto the fault emulator. Memory (dedicated block-RAM or external SRAM/DRAM) can be used as buffer for fault configurations to optimize communication between software and hardware components [LGPE07]. Unfortunately, especially when considering multiple faults, memory is likely to be exceeded by fault configurations and need to be updated frequently, causing again idle times in hardware. As a result, the communication between software components and the FPGA is still the performance limiting factor of fault emulation.

In this work, I address this issue with new performance optimizations. In Section 8.2 I focus on optimizations that remove the communication bottleneck while maintaining the flexibility to configure arbitrary multiple faults, and hence, close the gap between performance configurability.

5.4 Test Generation, Upload and Execution

The circuit altered for fault injection is sensitized by a test during fault emulation. As outlined in Section 3.4, a test sensitizes and propagates faults through a sensitized path, without which faults are not observable at primary or pseudo primary outputs. In the literature of emulation domains, a test is often also referred to as testbench [LGPE05] or workload [EMEM14]. Both structural and functional tests can be used to sensitize the emulated circuit.

In a security context, a test is used as representative of relevant situations, which is required to make use of to be verified hardware components by performing, e.g., encryption and decryption.

5.4.1 Structural Test

Structural tests can be generated using automated test pattern generation (ATPG) tools, available from industry and academia. To chose circuit input representing relevant situations in a security context, e.g., input of an hardware-implemented cryptographic algorithm [JHSS11], structural tests can be generated manually or using other strategies. However, for the application during security verification, structural tests are only applicable efficiently to module level, i.e. for security (sub-)components that are separated from complex architectures such as processor-based security designs in which these are usually integrated. After test generation, either a single test pattern is uploaded previously to performing corresponding fault experiments, or all tests are uploaded in advance and buffered in hardware.

Alternatively, in order to increase performance, structural tests can be automatically and autonomously generated on the FPGA. This completely removes communication overhead for test upload at the cost of acceptable hardware overhead for test generation. For instance, López-Ongil et al. [LGPE05] and Raik et al. [RETU05] use a linear feedback shift register (LFSR) for test generation.

5.4.2 Functional Test

Functional tests are usually used when the CUV executes code from memory, e.g., processor and microcontroller designs. To sensitize processor-based security designs, functional tests are usually built following a programming guidance and may additionally implement software-based fault countermeasures, hence, resulting in long and complex test sequences. Commercial tool-support is not available for automated test generation of functional tests [Reo15], and therefore, functional tests are written manually by a test engineer.

Once a functional test is built, it is loaded into the CUV's code memory, from where it is executed during fault experiments. An upload is required only once in advance because fault injection into the CUV usually does not corrupt code (read-only, except for architectures that support self-modifying code).

Important to note is that fault injection may affect the program's data flow and instruction sequence, e.g., by executing corrupted instructions or by executing corrupted jumps, which even may result in executing endless loops. Deviation of the data flow due to fault injection simply represents propagation of fault effects, which need to be propagated to observable locations. Deviation of the instruction sequence due to fault injection may also be observed this way. However, such fault effects constitute corrupted test execution. That is, in contrast to a structural test sequence, the test sequence of a functional test itself depends on fault behavior. This constitutes a serious issue, especially in case that faulty response observation and following fault classification depend on the timing of test execution. In this case, emulation results are unreliable, which possibly cause false positives during fault classification.

5.5 Response Observation and Fault Classification

As indicated in Figure 5.1, during fault emulation the CUV's response is observed in hardware on the FPGA. Following tasks and generation of fault-free references (golden reference) can be implemented in software as well. In general, it is preferable to implement fault classification, including generation of fault-free references, in hardware on the FPGA in order to minimize data exchange between software and hardware components. However, there are also approaches reported in literature that realize fault classification in software, e.g., [JHSS11].

After fault classification, the results need to be evaluated manually by a designer or verification engineer. Often, a logic simulation is necessary to track fault behavior to comprehend circuit behavior. This is a very time-consuming task for which knowledge of the CUV's architecture is required. It is therefore a necessity to prevent false positives in emulation results, allowing to use limited verification times efficiently.

Subsequently, approaches for response observation and fault classification reported in literature are reviewed.

5.5.1 Comparison of Two Circuit Instances

If fault emulation is used as accelerator for classical fault simulations, e.g., for determining fault coverage of manufacturing tests, fault emulation typically uses a second fault-free instance of the CUV as fault-free reference. Then, both fault-free and faulty emulations are executed in parallel. A comparator is used to observe and compare their responses at top level outputs of both CUV instances in each clock cycle of a given test. The comparator signalizes whether the fault is detected or not, which basically realizes a fault classification in detected (failure) and undetected (pass), as outlined in Section 3.2. For instance, Raik et al. [RETU05] and Ellervee et al. [ERTU07] present such an approach.

Comparing two CUV instances is easy to implement, however, it requires a lot of hardware resources on the FPGA. As a consequence, the circuit size for which fault emulation is applicable using this approach is limited because of limited hardware resources on FPGAs.

Since comparison is performed in a cycle accurate fashion, timing deviation in test execution caused by fault injection are not tolerated. This is similar to fault simulation, where faults are typically only categorized into detected and undetected faults for fault grading of manufacturing tests. In a security context, however, it is of importance to tolerate fault-induced timing deviation in test execution to some extend. For example, a fault could trigger an error correction measure that prolongs test execution just slightly by e.g. one single clock cycle, which constitutes uncritical desired behavior in the presence of faults. However, the fault would be observed at an output and is therefore classified as failure when, in fact, the service is actually delivered. Such cases need to be considered for response observation and fault classification in order to prevent that emulation results are bloated with false positives.

In conclusion, because of the generated hardware overhead and also because of the lack to tolerate fault-induced timing deviation of fault emulation, comparison of two CUV instances does not suit fault classification for security designs.

5.5.2 Alternative Approaches

Alternatively to using a second instance of the CUV, a fault-free emulation and a faulty emulation can be executed time-serially using the same CUV instance. Such an approach is proposed by e.g Shirazi et al. [SMS13]. While the hardware overhead is considerably reduced, the emulation is executed twice for each fault, once fault-free and once faulty. As a consequence, the performance is reduced by factor two.

Fault-free CUV responses can also be determined and stored as reference once in advance to executing fault experiments. Then, the stored reference can be used for fault classification either in the controlling software or on the FPGA. This approach provides the most advantages when the CUV generates a single, specific output per fault experiment. For example, this approach is suitable when output is transferred with a communication interface [NR11] or when output of a cryptographic algorithm is observed [JHSS11].

Note, approaches that are sensitive to specific output require to introduce an additional timeout to terminate the fault emulation in case that the CUV does not respond anymore due to fault injection [NR11, JHSS11, PSC^{+}12]. A timeout can also be used to determine whether a service, e.g. a safety-critical service, is performed in time. In a security context, however, a timeout does not necessarily point to a violated security requirement. For instance, the CUV may stall completely, and therefore, does not respond anymore. Although availability is violated, a stalled security device does not leak secrets.

5.5.3 Response Observation for Processor Designs

Processor designs usually provide only very limited top-level interfaces. Therefore, the typical concept of observing CUV responses at top-level is unattractive for processor designs. When top-level outputs are used for comparison of faulty and fault-free responses, the test executed by the processor needs to explicitly write to output ports in order to sensitize a path to these. This represents a performance bottleneck, especially when top-level ports are bound to specific protocols, e.g. UART or even more complex ones such as Ethernet. This motivates the demand for alternative techniques that suit processor designs. Furthermore, it is often required to get detailed information about faulty behavior in order to analyze possible issues in such complex designs [PSC^{+}12].

Sauer et al. [ST^{+}11] and Pellegrini et al. [PSC^{+}12] apply their FPGA-based fault emulator (circuit instrumentation) to microprocessors, namely OpenSparc T1 core and a MIPS-like processor respectively. Sauer et al. apply a functional test that stores calculated results in the processor's memory on the FPGA. Since this memory is in the address space of the CUV, undesired side-effects such as writing the result to a different location or overwriting results are likely to happen. Hence, dedicated memory should be used to store emulation results. Pellegrini et al. observe fault effects by using different fault detectors for e.g. traps, kernel panics and hardware stalls, partially based on observing the processor's program counter. This allows to track fault and circuit behavior and, in turn, allows to find design issues.

In the next Chapter 6 I propose a robust and efficient method for response observation and fault classification suitable for processor designs. It evaluates circuit behavior and observes fault effects based on the processor's program counter, which is supported by a functional test (software-based self-test) that propagates faults to internal observation points. This way, advantages of generating a single reference for fault classification apply, a second CUV instance is not required and fault-free and faulty responses are not generated time-serially.

5.6 Summary and Discussion

For the purpose of modeling fault attacks, the circuit instrumentation technique constitutes the most effective and most efficient technique. The complete circuit can be prepared for arbitrary multiple fault injection by performing a single circuit synthesis run. Software-based fault selection and generation provide the required flexibility for mimicking fault attacks based on, e.g., layout information, and hence, suits the purpose of emulating fault attacks the best. Therefore, the fault emulator, optimized for mimicking fault attacks, proposed next in Chapter 6 utilizes circuit instrumentation and software-based fault selection and generation.

In the following, I discuss in which directions I am going to improve existing work with the goal to emulate arbitrary fault attacks on security designs. The main focus lies on supporting the required configurability without performance loss compared to fault-free emulation. Moreover, the applicability is an important factor, which is closely entangled with hardware requirements because of limited hardware resources on FPGAs.

Applicability In Section 6.6, I present how fault classification can be realized without using a second instance of the CUV, saving hardware resources on the FPGA and increasing applicability for larger designs. For this purpose, a fault-free golden emulation is performed once in advance to performing fault experiments. The proposed method improves the approach that is sensitive to a single output by means of observing internal signals of the

CUV during fault experiments. These are used to monitor a manageable set of design- and test-related events. This allows to tolerate fault-induced timing deviation during fault emulation and to identify timeouts caused by hardware stalls caused by fault injection to reduce false positives in emulation results. Fault classification for security designs is realized by introducing two sets of observation points, one to observe the functional behavior of the verified circuit and one to observe the effectiveness of fault countermeasures.

As I am going to detail in Chapter 7, there are issues when instrumenting combinational cells in hardware, e.g. high hardware overhead and issues fixing timing constraints. Therefore, for fault injection in combinational logic I propose in Chapter 7 to extend fault emulation by a software-based method in order to avoid the instrumentation of combinational cells. This way, the applicability of emulation-based fault injection in combinational cells is enhanced in terms of hardware requirements and the issues w.r.t. to fixing timing constraints are eliminated.

In contrast to available alternatives, namely software- and simulation-based fault modeling methods, once fault configurations are transferred to the FPGA, the runtime for fault emulation equals the test duration, i.e. the duration of the workload. This corresponds to a constant complexity and enables real time test execution. Hence, fault emulation suits long functional tests, and in fact, benefits from these as I am going to show with experimental results in Chapter 9, whereas software-based approaches struggle.

Configurability In contrast to approaches that generate fault configurations autonomously in hardware on the FPGA, the proposed fault emulation environment implements fault selection and fault generation in software. This enables to configure arbitrary faults, for which all fault properties are configurable at runtime.

Performance of Fault Emulation The performance of fault emulation depends on the operating clock frequency applied to the synthesized CUV. Since additional logic for fault injection is inserted into combinational paths, it might be necessary to reduce the clock frequency to fix setup time violations. This is usually negligible for fault injection into FFs since just a few additional cells are inserted once per FF, i.e. once in a critical combinational path. In case of instrumenting combinational cells, the impact on performance is usually less than an order of magnitude. For instance, a clock divider of five is used in [PSC$^+$12] for a SPARC-V9 architecture. Hence, this impact can also be neglected in comparison to software-based alternatives, since fault emulation is three to five orders of magnitude faster

[EMEM14, EVC+09].

Because of data exchange between software and hardware components, the runtime for fault emulation develops in the worst case linearly with the data throughput of the communication interface, as I am going to show in Chapter 9. Configuring multiple faults generates more data, which is frequently transferred to the FPGA. I am also going to show later that this causes idle times in hardware that easily exceed many times the net test duration of the executed workload. Hence, this aspect poses a considerable performance bottleneck, and therefore, I tackle this issue by proposing appropriate optimization measures in Chapter 8. These allow to reduce the runtime of fault emulation such that it equals the duration of the sensitizing test.

Chapter 6

Fault Emulation Environment for Security Verification

In this chapter I am going to present the FPGA-based fault emulation environment that I propose for modeling fault attacks during security verification. I presented a fault emulation environment in [NR11] that I implemented for my Master's thesis, which therefore does not contribute to this thesis. In [NR11] I also published an approach for modeling multiple bit-flip faults in sequential logic using synchronous fault injection cells, which was not part of my Master's thesis and contributes to this thesis. I published a major rework of these concepts including refinements of basically all aspects involved by the implemented fault emulation environment in [NHN^{+}14, NHRS15]. The formal description of the implemented gate level fault model is published in this thesis for the first time.

The fault emulation environment that I propose for modeling fault attacks is illustrated in Figure 6.1. It is implemented using the circuit instrumentation technique. This allows arbitrary multiple fault injection in combinational and sequential logic. Furthermore, this fault injection methodology guarantees high performance and requires only a single synthesis run to prepare the complete circuit for fault injection.

The fault emulation environment consists of a software part executed by a host computer (depicted on the left of Figure 6.1) and a hardware part implemented using an FPGA (depicted on the right of Figure 6.1). The software initiates and controls the fault emulation and realizes fault generation, upload of fault configurations and test upload. This provides maximum flexibility for fault selection and fault generation, and therefore, suits sophisticated fault selection strategies based on, e.g., layout information. A communication interface (UART in Figure 6.1) connects the host computer with an FPGA-board.

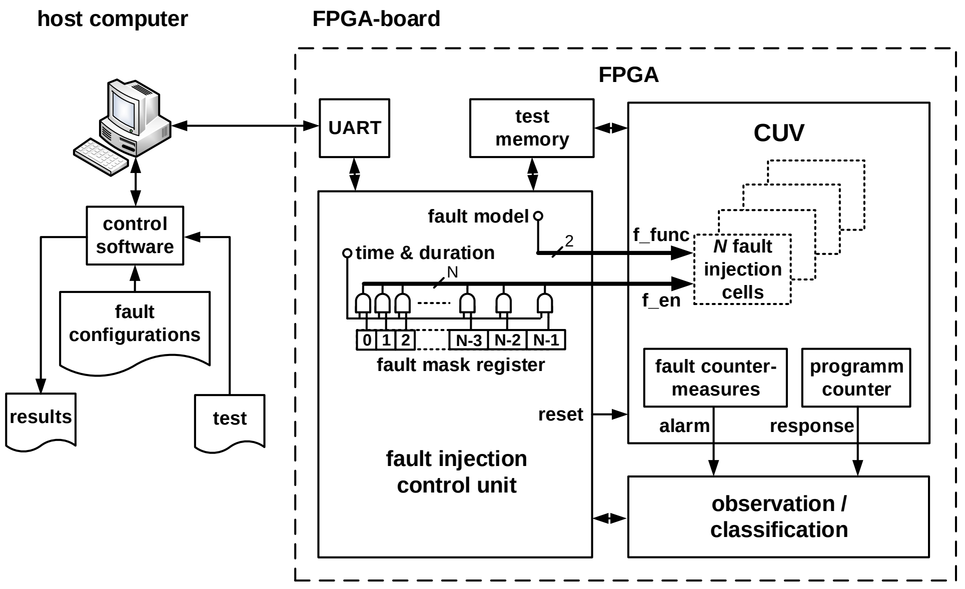


Figure 6.1: The fault emulation environment utilizing a host computer and an FPGA-board. Note that N corresponds to N_L.

Next, in Section 6.1, a gate level fault configuration model is presented, which is described by means of refining properties of the meta configuration fault model introduced in Chapter 4. Section 6.2 describes the hardware components that implement the gate level fault configuration model and controls fault injection into the circuit under verification (CUV). Then, in Section 6.3, I present a completely automated synthesis setup, which instruments a CUV for fault injection and maps the instrumented CUV as well as the controlling hardware components onto an FPGA. After this, in Section 6.4, I discuss how fault injection campaigns are realized using the presented fault emulation environment. In Section 6.5, I present a new efficient method for response observation, allowing to interpret emulation results for security designs in a robust and efficient way, which constitutes the base for fault classification of security designs, which I present in Section 6.6.

6.1 Fault Configuration Model at Gate Level

The fault emulation environment can be applied to synchronous sequential circuits, which can be represented as finite state machine. I therefore give a brief description of finite state machines, which is then used as base to apply the fault configuration model presented in Chapter 4 at gate level. The resulting gate level fault configuration model is implemented for fault injection in sequential and combinational logic using FPGA-based fault emulation, presented next in Section 6.2. Chapter 7 implements the gate level fault configuration model for combinational logic using a software-based method, which is then combined with FPGA-based fault emulation to accelerate sequential fault propagation.

6.1.1 Finite State Machine

The following description is mainly inline with [KKJ10]. Finite state machines (FSM) describe a finite number of internal states and state transitions to model the input-output behavior of software and digital circuits. The FSM is defined as tuple $M = (I, O, S, s_0, \delta, \lambda)$, which includes an input alphabet I, an output alphabet O representing the external state, a set of internal states S, a transition (next-state) function δ and an output function λ. The FSM starts from an initial state $s_0 \in S$ in which an initial input is applied. At any discrete instance of time t, the FSM is in a defined current state. The transition function $\delta : S \times I \rightarrow S$ determines the state transition to the next state dependent on the current state and the current input. The output function λ determines the output of the FSM either dependent on only the current state (Moore $\lambda : S \rightarrow O$) or dependent on both the current state and the current input (Mealy $\lambda : S \times I \rightarrow O$). The FSM generates a finite next-state sequence and a finite output sequence in response to a finite input sequence.

A typical hardware implementation of an FSM is depicted in Figure 6.2, which is composed of sequential cells (depicted as positive edge-triggered DFFs) representing the state s[t] of the circuit and combinational logic cells (depicted as *comb*), realizing the state transition function δ and output function λ. Combinational and sequential cells are connected with logic lines in a netlist, which transport cell's logical output from one cell to other cell's inputs. In each state $s[t]$, the combinational logic generates the respective next-state $s[t+1]$ which is latched by sequential cells at the discrete time $t+1$. Discrete times correspond to clock edges of the circuit's clock signal. The circuit remains in a state until the next sensitive clock edge arrives at sequential cells. That is, the state is active for the entire clock cycle if either positive or negative edge sensitive sequential cells are used. If both types of sequential cells are combined, then the state is active for half a clock cycle. Note, in the remainder of this work it is assumed that one type of sequential cells is used exclusively. Input patterns x are binary representatives of the input alphabet I, which are applied at primary inputs (PI). Analogously, input patterns of the combinational circuit part, denoted by $x_C[t]$, are applied

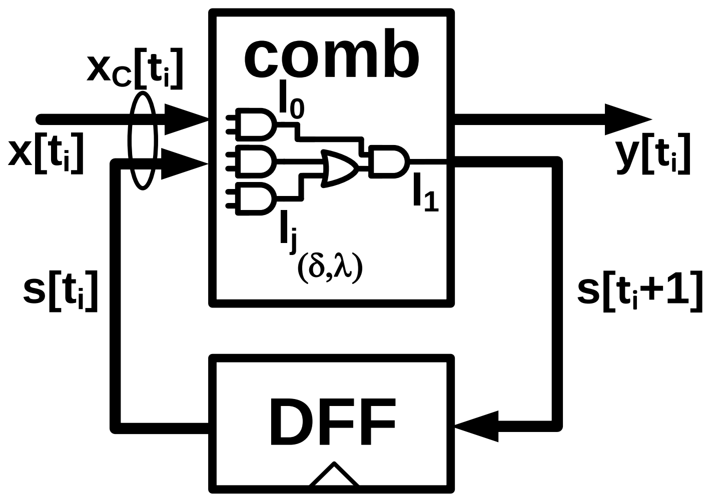


Figure 6.2: FSM, composed of DFFs and combinational logic (comb). Logic lines $L = \{ l_0, \dots, l_{j-1} \}$ that are connected to combinational cell's outputs are indicated. l_1 is exemplary connected to data inputs of DFFs.

at pseudo primary inputs (PPI). The outputs y are binary representatives of the output alphabet O, which are driven at primary outputs (PO).

6.1.2 Fault Configuration Model at Gate Level

Subsequently, I refine properties of the meta fault configuration model presented in Chapter 4 for gate level. This way, I derive the fault configuration model that I am going to implement at gate level in this chapter. This fault configuration model covers single and multiple permanent and transient faults in combinational and sequential logic (SEU, MEU, SET, MET). For transient faults, a cycle accurate fault model similar to the one for single transient faults presented in [PHRB11] is used. That is, fault injections as well as releasing faults are aligned with the sensitive clock edge and only logical masking is considered (refer to Section 3.5). In contrast to [PHRB11], stuck-at, set/reset and bit-flip fault model types and multiple faults are considered, for which a single fault injection time (number of fault injection times $n = 1$) and the fault duration are configurable. Note that this allows to model physical fault attacks that inject faults at a single point in time. Modeling more advanced fault attacks that exploit multiple fault injection times, such as second order attacks using time-displaced laser shots, are enabled with an optimization that I am going to propose in Section 8.2.2. Although, the implemented fault configuration model is implemented at gate level, note that fault injection in flip flops also suits fault injection at RT level.

First, I define the spatial properties of the implemented fault model by means of defining fault injection locations for combinational and sequential logic. Next, the value configuration space and the temporal properties of the fault model are specified. After this, I join the specified properties by defining appropriate forcing functions and configuration functions allowing to configure bit-flip, stuck-at and set/reset faults, where also the respective subsets for single faults and permanent faults are discussed.

Refining Spatial Properties Cell's input and output pins and logic lines, i.e. nets connecting pins, constitute adequate fault injection locations at gate level. Note that a fault modeled for a cell's output pin is meant to model a fault in the output circuitry of a cell and affects the entire connected net of logic lines as well as the input pins connected to it. Hence, a fault modeled for an output pin is equivalent to a fault modeled for the connected net of logic lines and it is equivalent to a multiple fault modeled for all connected input pins. In contrast, a fault modeled for a cell's input pin is meant to model a fault in the input circuitry of a cell without coupling the fault into the connected net. That is, faults modeled for input pins do not affect the connected net of logic lines. In case that a fault modeled for an input pin is propagated through the respective cell, there exists an equivalent fault for the output pins of this cell. As a consequence, it would be sufficient to model faults for output pins only. For combinational logic I therefore decided to model faults for output pins, however, for sequential logic I model faults for input pins. Modeling faults at inputs of sequential cells has the advantage that faults at the bound from combinational to sequential logic can be described, which especially helps to determine fault equivalences of faults in combinational and sequential logic. Determining fault equivalence constitutes the base for extending fault emulation by a software-based pre-processing for enhancing fault injection in combinational logic, which is further detailed in Chapter 7. If outputs of sequential cells had been chosen as fault injection locations, then the time delay of sequential cells would have been required to be considered when mapping faults from combinational logic to equivalent faults in sequential logic. Strictly speaking, this would already correspond to modeling the circuit behavior, which I keep separated from the fault configuration model. Based on these considerations, the Definition 4.1.3 for fault injection locations is refined for gate level as follows:

Fault injection locations for combinational logic, denoted by $l_j \in L_{\mathrm{C}}$, are now specified as the set of output pins of computational cells, so

$$L_{\mathrm{C}} = \{\, l_j \mid l_j \text{ is an output pin of a combinational cell} \,\}\,, \tag{6.1}$$

where L_{C} is a subset of the set of all fault injection locations L, so $L_{\mathrm{C}} \subseteq L$. This is, for combinational logic, faults are injected at output pins, affecting the connected net of logic lines.

The set of fault injection locations for sequential logic, denoted by $l_j \in L_{\mathrm{FF}}$, is defined as the set of input pins of sequential cells. So,

$$L_{\mathrm{FF}} = \{\, l_j \mid l_j \text{ is a data input pin of FFs} \,\}\,, \tag{6.2}$$

where $L_{\mathrm{FF}} \subseteq L$.

The union of both sets, L_{C} and L_{FF}, is the set of all fault injection locations L, so $L_{\mathrm{C}} \cup L_{\mathrm{FF}} = L$.

Refining Temporal Properties With respect to the state machine depicted in Figure 6.2, clock cycles correspond to the time interval for which a circuit remains in a state. Based on this consideration, the Definition 4.1.1 for time intervals and related temporal properties are further specified for gate level as follows: The shortest time interval $T_i \in T$ for which fault injection can be performed is chosen to be one clock cycle. Then, it is sufficient to specify T_i as subset of $\mathbb{N}_0$ (instead of $\mathbb{R}_0$), so $T_i \subset \mathbb{N}_0$, where $T_i = [t_i, t_{i+1})$ and $t_i, t_{i+1} \in \mathbb{N}_0$. It is sufficient to consider only clock cycles in which the circuit under consideration is sensitized by a test. At the sensitive edge of a clock cycle the circuit can transition into an other state, so $s[t_i] \to s[t_{i+1}]$. Therefore, the set of time intervals T is now refined respectively. So

$$T = \{\, T_i \mid [t_i, t_{i+1}) \subset \mathbb{N}_0 \text{ and } T_i \text{ is a clock cycle} \,\}\,, \tag{6.3}$$

where t_i is the discrete time when state transitions $s[t_i] \to s[t_{i+1}]$ occur.

Since t_i and t_{i+1} are discrete consecutive times, where $t_i, t_{i+1} \in \mathbb{N}_0$, the discrete time t_i is the only element in the time interval $T_i = [t_i, t_{i+1})$, so $[t_i, t_{i+1}) = \{t_i\}$. Hence, the fault injection time t_{inj} and the fault release time t_{release} can be clearly used as a synonym for their respective intervals in which they are included. Note that this is similar to using a time t instead of a time interval when describing a state $s[t]$ of a state machine (see also Section 6.1.1).

According to Definition 4.3.3, the fault duration d defines the number of consecutive time intervals, i.e. clock cycles which map to the same forcing function $\varphi \neq \varphi_u$. This is, for the fault duration d the same fault injection locations are affected, and all $l_j \in L_a$ map to an individual value $v \neq u$. The fault duration is at least one cycle ($d \geq 1$).

Specifying the Value Configuration Space The implemented fault model at gate level covers a value configuration space that allows to configure the fault models stuck-at-0, stuck-at-1, reset, set and bit-flip. Therefore, the Definition 4.1.7 for the value configuration space is refined for gate level as follows:

$$V_u = \{ 0, 1, b, h, u \} , \tag{6.4}$$

where $v \in V_u$ and

- $v = 0$: force logic level zero, used for stuck-at-0 and reset faults,
- $v = 1$: force logic level one, used for stuck-at-1 and set faults,
- $v = b$: force inverted logic level of fault-free fault injection location l_j, used for bit-flip faults,
- $v = h$: hold logic level of fault injection location l_j of previous time interval T_{i-1}, used for fault duration
- $v = u$: unaffected (fault-free) according to Definition 4.1.10.

If l_j maps to $v = h$, then l_j is forced the logic level of l_j that was driven in the previous time interval T_{i-1}, where $T_{i-1} = [t_{i-1}, t_i)$ and $T_{i-1} \in T$. This allows to configure the sequence of inverting the logic level of a fault injection location l_j (bit-flip with $v = b$) for one clock cycle followed by holding this initially forced logic level (hold with $v = h$) for additional $d - 1$ clock cycles as it would be a temporary stuck-at (set/reset) fault. Such a configuration is necessary for bit-flip faults when considering fault durations $d > 1$. Without the ability to hold the same faulty logic level for $d - 1$ clock cycles the resulting faulty logic level would toggle dependent on the original fault-free logic level, which does not reflect actual physical fault behavior in a realistic way.

6.1.3 Multiple Transient and Multiple Permanent Faults

Subsequently, the family of fault configuration functions that allows to configure stuck-at, set/reset and bit-flip faults is specified, where single fault injection times and configurable fault durations are considered. Note, according to the fault configuration model, permanent faults just constitute the special case where $t_{\text{inj}} = t_0$ and $t_{\text{release}} = t_{N_T}$. Therefore, the following definitions include both permanent and transient faults.

The relevant subsets of the value configuration space $V_u = \{ 0, 1, b, h, u \}$ for stuck-at and bit-flip faults when considering single fault injection times and configurable fault durations are denoted by V_{SA} and V_{BF} respectively, where

$$\begin{gathered} V_{\mathrm{SA}} = \{\, 0, 1, u \,\}\,,\ v \in V_{\mathrm{SA}} \\ \text{and} \\ V_{\mathrm{BF}} = \{\, b, u \,\}\,,\ v \in V_{\mathrm{BF}}. \end{gathered} \tag{6.5}$$

In order to hold the forced values, required to realize fault durations, the subset V_{H} is defined, where

$$V_{\mathrm{H}} = \{\, h, u \,\}\,,\ v \in V_{\mathrm{H}}. \tag{6.6}$$

Note, in order to model unaffected fault injection locations, all value configuration spaces include the unaffected value u.

These specific value configuration spaces allow to define families of forcing functions, i.e. subspaces of the spatial-value configuration space ${V_u}^L$, dedicated to the purpose of either injecting stuck-at faults, injecting bit-flip faults or holding the forced value, denoted by Φ_{SA}, Φ_{BF} and Φ_{H}, respectively, where

$$\begin{aligned} \Phi_{\mathrm{SA}} &= \left\{\, \varphi \;\middle|\; \varphi \in (V_{\mathrm{SA}})^L \,\right\},\ \varphi_{\mathrm{SA}} \in \Phi_{\mathrm{SA}}, \\ \Phi_{\mathrm{BF}} &= \left\{\, \varphi \;\middle|\; \varphi \in (V_{\mathrm{BF}})^L \,\right\},\ \varphi_{\mathrm{BF}} \in \Phi_{\mathrm{BF}}, \\ \Phi_{\mathrm{H}} &= \left\{\, \varphi \;\middle|\; \varphi \in (V_{\mathrm{H}})^L \,\right\},\ \varphi_{\mathrm{H}} \in \Phi_{\mathrm{H}}, \end{aligned} \tag{6.7}$$

where $\varphi \in {V_u}^L$ and $(V_{\mathrm{SA}})^L, (V_{\mathrm{BF}})^L, (V_{\mathrm{H}})^L \subset {V_u}^L$.

Finally, analogously to the fault configuration function for single fault injection times (Definition 4.3.9), these forcing functions are used to define a family of fault configuration functions for stuck-at faults, denoted by f_{SA}, and bit-flip faults, denoted by f_{BF}, where $\mathrm{SA} = (\varphi'_{\mathrm{SA}}, T_d)$ and $\mathrm{BF} = (\varphi'_{\mathrm{BF}}, T_d)$ and $T_d = [t_{\mathrm{inj}}, t_{\mathrm{release}})$ and $T_d \subseteq T$ and $t_{\mathrm{inj}}, t_{\mathrm{release}} \in \mathbb{N}_0$ and $t_{\mathrm{inj}} < t_{\mathrm{release}}$. A single fault injection time t_{inj} and fault durations $d = t_{\mathrm{release}} - t_{\mathrm{inj}}$ are configurable.

$$\begin{gathered} f_{\mathrm{SA}}(T_i) = \begin{cases} \varphi'_{\mathrm{SA}} & \text{for } T_i = T_{\mathrm{inj}} \\ \varphi'_{\mathrm{H}} & \text{for } T_i \in (T_d \setminus T_{\mathrm{inj}}) \\ \varphi_u & \text{otherwise} \end{cases} \\ \text{and} \\ L_a\left(\varphi'_{\mathrm{SA}}\right) = L_a\left(\varphi'_{\mathrm{H}}\right) \end{gathered} \tag{6.8}$$

$$\begin{gathered} f_{\mathrm{BF}}(T_i) = \begin{cases} \varphi'_{\mathrm{BF}} & \text{for } T_i = T_{\mathrm{inj}} \\ \varphi'_{\mathrm{H}} & \text{for } T_i \in (T_d \setminus T_{\mathrm{inj}}) \\ \varphi_u & \text{otherwise} \end{cases} \\ \text{and} \\ L_a\left(\varphi'_{\mathrm{BF}}\right) = L_a\left(\varphi'_{\mathrm{H}}\right), \end{gathered} \tag{6.9}$$

where $f_{\text{SA}},\ f_{\text{BF}} \in \left(V_u{}^L\right)^T$ and $T_i \in T$ and $T_i = [t_i, t_{i+1})$ and $t_i \in \mathbb{N}_0$. These fault configuration functions define a single injection interval $T_{\text{inj}} = [t_i, t_{i+1})$, where $t_i = t_{\text{inj}}$, that maps to a particular forcing function φ'_{SA} for stuck-at faults and to a particular forcing function φ'_{BF} for bit-flip faults. In case of stuck-at fault injection, affected fault injection locations $l_j \in L_a$ map individually to either $v = 0$ or $v = 1$, which allows to combine stuck-at-0 and stuck-at-1 faults when configuring multiple faults. In case of bit-flip fault injection, affected fault injection locations $l_j \in L_a$ map to $v = b$. Note that all $l_j \notin L_a$ map to $v = u$, so these are unaffected.

Since a single fault injection time is described, the time intervals $T_d \setminus T_{\text{inj}} = [t_{i+1}, t_{\text{release}})$ map to the respective φ'_{H}. Note that for stuck-at faults $L_a\left(\varphi'_{\text{SA}}\right) = L_a\left(\varphi'_{\text{H}}\right)$ and for bit-flip faults $L_a\left(\varphi'_{\text{BF}}\right) = L_a\left(\varphi'_{\text{H}}\right)$. This describes that after mapping fault injection locations $l_j \in L_a$ to individual values $v \neq u$ fault injection is being held for the same fault injection locations.

All other time intervals map to the fault-free forcing function φ_u, so in all other time intervals all fault injection locations map to the fault-free value $v = u$. This way, a configuration sequence is defined which either results in a stuck-at or a bit-flip fault with a configurable fault duration. Although the ability to hold the same value is only necessary for bit-flip faults, it is defined this way for both stuck-at and bit-flip faults in order to allow a generic implementation in hardware.

If the fault duration is less than $t_{N_T} - t_0$, so $d < t_{N_T} - t_0$, then a transient fault is configured, otherwise a permanent fault is configured. Temporary stuck-at-0 and stuck-at-1 faults correspond to reset and set faults, respectively. The spatial fault multiplicity m determines whether a single or multiple fault is configured, according to the discussion in Section 4.2.1. Single faults and permanent faults are further discussed subsequently.

In the remainder of this work the short description of the fault configuration functions is used, i.e. the notation for a particular parametrized fault according to Definition 4.3.8. For example, a particular bit-flip fault is denoted by BF_1, where $\text{BF}_1 = (\varphi'_{\text{BF}}, T_d)$, $T_d = [t_{\text{inj}}, t_{\text{release}})$ and $\varphi'_{\text{BF}} \in \Phi_{\text{BF}}$ is a particular forcing function for bit-flip faults.

6.1.4 Single Transient and Single Permanent Faults

According to the discussion in Sections 4.2.1 and 4.4.1, single faults describe the m-th subset of $\mathcal{F}_{\text{spatial}}$ where $m = 1$. With respect to the presented gate level fault configuration model, a single stuck-at fault considering single fault injection times and configurable fault durations is defined according to Equation 6.8 when

$$|L_a\left(\varphi_{\text{SA}}\right)| = |L_a\left(\varphi_{\text{H}}\right)| = 1. \tag{6.10}$$

A single bit-flip fault considering single fault injection times and configurable fault durations is defined according to Equation 6.9 when

$$|L_a(\varphi_{\mathrm{BF}})| = |L_a(\varphi_{\mathrm{H}})| = 1. \tag{6.11}$$

If the fault duration is less than $t_{N_T} - t_0$, so $d < t_{N_T} - t_0$, then a single transient fault is configured, otherwise a single permanent fault is configured. Single temporary stuck-at-0 and stuck-at-1 faults correspond to single reset and set faults, respectively.

6.2 Hardware Implementation

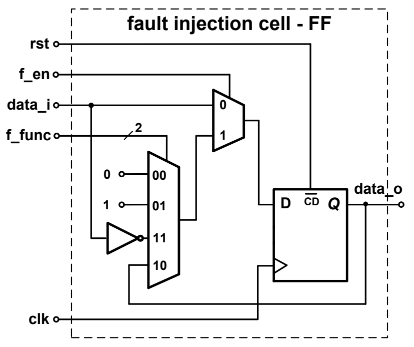
As depicted earlier in Figure 6.1, the *Circuit Under Verification (CUV)*, e.g. a security controller, a *fault injection control unit*, an *observation / classification* unit, a *UART* interface and a *test memory* are loaded into an FPGA (depicted as a dashed rectangle). Faults, configured by the *fault injection control unit* according to fault configurations received over the UART interface, can be injected at fault injection locations, which are either inputs of FFs or combinational cells' outputs located in the CUV. The UART is also used to transmit the test to be executed during fault experiments into the test memory.

The observation / classification unit is used to monitor the response of the CUV in presence of faults. It also generates the fault-free reference, which is used for comparison versus faulty responses to allow fault classification. Contrarily to the conventional approaches that uses two instances of the CUV, the CUV is instantiated only once to reduce hardware overhead. The fault-free reference is generated by performing a single fault-free emulation (golden run) in advance, and its result is used for all following fault experiments.

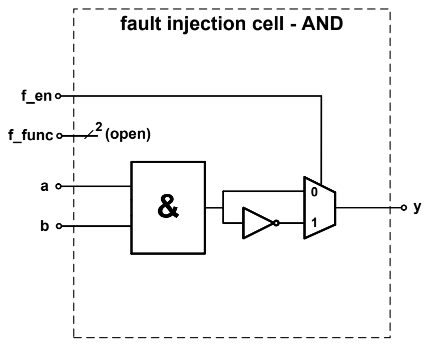
Response observation and fault classification are further outlined in Sections 6.5 and 6.6. In the remainder of this section the hardware components of the fault emulation environment that implement the gate level fault configuration model, namely the fault injection control unit and fault injection cells for combinational and sequential cells implementing fault injection locations, are presented.

6.2.1 Fault Injection Control Unit

The fault injection control unit makes use of a *fault mask register* in which each bit determines whether a fault is injected at an associated fault injection location $l_j \in L$. Bits of the *fault mask register* are addressed by their index



(a) instrumented flip flop



(b) instrumented AND gate

Figure 6.3: Fault injection cells replace original FFs and combinational cells in the CUV. (a) Fault injection cell for sequential cells. (b) Fault injection cell for combinational cells with an example of a two-input AND gate.

$j \in \{0, \ldots, N_L - 1\}$, are sorted in ascending order and can be set separately for multiple fault selection. The number of its bits $N_L = |L|$ corresponds to the spatial granularity of the implemented fault model, which is consistent with the discussion before.

In case that a bit in the *fault mask register* is set to logical '0', fault injection is disabled at respective fault injection locations and the *fault mask register* implements the assignment $l_j \longmapsto u$, where $u \in V_u$ according to Equation 6.4. In the other case that a bit is set to logical '1', a value assignment $l_j \longmapsto v$ is selected and $v \neq u$, where $v \in V_u \setminus u$. The actually forced value is then determined by the *fault model* type according to the implementation of fault injection cells, as explained in the next section.

Incremental selection of multiple fault injection locations in the *fault mask register*, i.e. setting multiple of its bits to logical '1', results in a respective multiple fault.

6.2.2 Fault Injection Cells

The CUV's fault injection locations include FFs or combinational cells, which are replaced by fault injection cells during circuit instrumentation in order to provide fault injection capability. This replacement is done during synthesis, which is detailed in Section 6.3. Basically, fault injection cells constitute the hardware implementation of the fault model's value configuration space and temporal activation of fault injection realizes the temporal properties of the implemented fault model. A fault injection cell for FFs is depicted in Figure 6.3a and one fault injection cell for combinational cells with an example of a two-input AND gate is depicted in Figure 6.3b.

Fault injection cells for FFs can be configured to either represent a bit-flip, stuck-at-0 or stuck-at-1 fault. The fault duration and fault injection time can be configured as well. This allows to emulate bit-flip faults which hold the initial forced value for multiple clock cycles. Note that bit-flip faults always invert the value of a fault injection location, i.e. bit-flip faults always cause errors, whereas stuck-at faults do not manifest as error without sensitization. Furthermore, temporary stuck-at faults, also known as set and reset faults, can be configured. This is for example useful to mimic fault effects at asynchronous pins and to mimic physical fault effects in a less pessimistic fashion, compared to bit-flip faults. A configurable fault duration allows to model e.g. long duration laser-induced faults at gate level or RTL, which were recently investigated at transistor level [LDCDN+15].

Fault Injection in FFs The signal *f_en* of each *fault injection cell* is associated with an individual bit of the *fault mask register* (sorted in ascending order). If the corresponding bit in the fault mask register is set to logical '1' and the configured fault injection time $t_{\text{inj}} \in T_{\text{inj}}$ is reached, then f_en is activated for a duration d, and therefore, selects a faulty behavior of the associated fault injection cell for the duration d. This is realized by an additional AND gate per fault injection cell, as depicted in Figure 6.1.

As long as f_en = '1', the control signals *f_func* select one of four possible fault model types, represented by the value assignment $l_j \longmapsto v$, where $v \in V_u \setminus u = \{\, 0, 1, b, h \,\}$. These value assignments select the fault model types stuck-at-0, stuck-at-1, flip and hold, respectively.

In case of a configured $d > 1$, the fault injection control unit configures a sequence of assigning a $v \in \{0, 1, b\}$ for one clock cycle and assigning $v = h$ for the remaining duration of $d - 1$ clock cycles. Holding the value assignment is implemented by the back coupling from the Q-pin to the D-pin of the FF depicted in Figure 6.3a. This is necessary for bit-flip faults in order to guarantee that the forced value is flipped only once. Otherwise it would flip whenever the data input changes its value.

The just discussed configuration possibilities together with fault-free configuration, which is realized by the fault injection control unit, implement the fault configuration functions for stuck-at faults f_{SA} and bit-flip faults f_{BF}, where a single fault injection time and variable fault durations are configurable, as defined in Equations 6.8 and 6.9, respectively.

In Figure 6.1, the width of f_func is depicted as a bus of 2 lines. These are controlled by two FFs, which are shared for all fault injection cells. Note that this setup does not allow to configure individual fault model types for each fault injection location, which is required for combinations of stuck-at-0 and stuck-at-1 faults when configuring multiple stuck-at faults. In order to allow combinations of stuck-at-0 and stuck-at-1 faults, a second setup is supported that enables the fault injection control unit to individually configure the function of each fault injection cell. For this purpose, a bus of $2 \cdot N_L$ lines is controlled by two individual FFs per fault injection cell, which completes the implementation of the forcing function φ_{SA}.

Fault Injection in Combinational Cells The applicability of FPGA-based fault injection in combinational cells is limited, mainly due to limited hardware resources on FPGAs. This aspect is discussed in detail in the next Chapter 7, where I tackle this issue with a methodology that maps faults from combinational logic to equivalent faults in sequential logic, utilizing a software-based pre-processing. However, the conventional emulation-based approach similar to the one presented in [PHRB11] that I expected to generate the least hardware overhead is outlined briefly. This setup is used later to benchmark the method proposed in Chapter 7 for enhancing fault injection in combinational logic.

As depicted in Figure 6.3b, fault injection cells for combinational cells are kept very simple. This is necessary to reduce hardware overhead and to minimize the additional logic for circuit instrumentation inserted in combinational paths. Consequently, this implementation is only able to inject bit-flip faults, and hence, the implemented value configuration space for combinational cells, denoted by V_{C}, is limited to $V_{\mathrm{C}} \subset V_u$, where $V_{\mathrm{C}} = \{\, b, u \,\}$. Furthermore, since back coupling is not implemented, which would implement the hold functionality, the fault release time is fixed to $t_{\mathrm{release}} = t_{\mathrm{inj}} + 1$. So, a fault duration of exactly one clock cycle ($d = 1$) is realized, which is controlled by assigning *f_en* to '1' for one clock cycle.

In the remainder of this chapter, fault injection in FFs is considered solely. Fault injection in combinational logic is further discussed in the next Chapter 7.

6.2.3 Concrete Fault Configuration Possibilities

In this section, I briefly outline concrete configuration examples covering all configuration possibilities of the implemented gate level fault configuration model. These cover the fault configuration functions φ_{SA} and φ_{BF} defined in Section 6.1.

Example 6.2.1 (Bit-flip fault, $\boldsymbol{d = 1}$)**.** A standard bit-flip fault is realized by inverting the data input of the FF (*f_func* = '11') while fault injection is enabled (*f_en* = '1') for exactly one clock cycle when the fault injection time $t_{\text{inj}} \in T_{\text{inj}}$ is reached.

Example 6.2.2 (Bit-flip-hold fault, $\boldsymbol{d \leq N_T}$)**.** A bit-flip-hold fault for the duration of $d \leq N_T$ clock cycles is realized by enabling fault injection (*f_en* = '1') for the duration d when the fault injection time $t_{\text{inj}} \in T_{\text{inj}}$ is reached. During this, the configuration sequence of inverting the data input of the FF (*f_func* = '11') for the first clock cycle followed by holding the faulty value (*f_func* = '10') for a configurable number of $x = d - 1$ clock cycles is assigned.

Example 6.2.3 (Reset/set fault, $\boldsymbol{d < N_T}$)**.** Analogously to the example for bit-flip-hold faults, it is also possible to configure a reset or set fault for a duration of $d < N_T$ clock cycles. It is realized by enabling fault injection (*f_en* = '1') for the duration d when the fault injection time $t_{\text{inj}} \in T_{\text{inj}}$ is reached. During this, the configuration sequence of forcing the data input of the FF for one clock cycle to '0' (*f_func* = '00') for reset faults or to '1' (*f_func* = '01') for set faults, followed by holding (*f_func* = '10') the faulty value for a configurable number of $x = d - 1$ clock cycles is assigned.

Example 6.2.4 (Stuck-at-0/1 fault, $\boldsymbol{d = N_T}$)**.** Permanent stuck-at-0 and stuck-at-1 faults are realized analogously to reset and set faults, however, with $t_{\text{inj}} = t_0$ and $t_{\text{release}} = t_{N_T}$, and therefore, $d = N_T$. Hence, stuck-at-0/1 faults are realized by the sequence of forcing the data input to '0' (*f_func* = '00') or to '1' (*f_func* = '01') during the very first cycle $t_0 \in T_{\text{inj}}$, respectively, followed by holding (*f_func* = '10') the faulty value until the emulation is terminated $t_{\text{release}} = t_{N_T}$. Additionally, the fault injection is enabled (*f_en* = '1') for the entire emulation ($d = N_T$).

6.3 Circuit Instrumentation

The presented fault emulation environment uses commercial synthesis tools to implement circuit instrumentation. During this process, combinational or sequential cells (fault injection locations) are replaced by a representation that adds fault injection capability, the fault injection cells presented earlier in Section 6.2.2.

There would be two alternative approaches, namely direct modifications in gate level netlists, e.g., presented in [PSC+12], or library modifications, e.g., presented in [JHSS11]. However, the former approach implements tasks that are already integrated in synthesis tools, e.g. design parsing and removing and creating cells. Using synthesis tools instead, powerful build-in synthesis commands are already available, which can be called from a synthesis script to automate circuit instrumentation. The latter approach uses a modified library, in which all FFs are replaced by cells that add fault injection capability. As a consequence, the library needs to be maintained whenever it is updated because of technology changes. Furthermore, using a modified library, all cells in a CUV are instrumented by means of mapping the design to the modified library. However, often it is necessary to exclude specific cells or entire design hierarchies from circuit instrumentation. This is not only useful to restrict fault injection to specific security relevant components, e.g. hardware implemented cryptographic algorithms. Often response observation of the CUV depends on debug or monitoring logic. In this case fault injection has to be prevented in these circuit parts since it may falsify emulation results.

The method I am going to present is completely based on build-in commands of synthesis tools, does not require direct netlist or library modifications and provides good flexibility by means of controlling circuit instrumentation.

Selection of Fault Injection Locations First, the CUV's design files are analyzed and elaborated using a synthesis tool, e.g. Synopsys Design Compiler. In a second step, either combinational or sequential cells are selected for circuit instrumentation. This step, excludes specific cells or entire design hierarchies from circuit instrumentation. This is automated by means of executing a configuration script.

Circuit Instrumentation In the next step, selected fault injection locations are replaced by a black box of the appropriate fault injection cell. For instance, any selected flip flop is replaced by a black box of the fault injection cell depicted in Figure 6.3a. For this purpose, build-in synthesis commands provided by a synthesis tool are used to remove cells, instantiate fault injection cells and reconnect these in the design. Moreover, the controlling signals *f_en* and *f_func* of fault injection cells are wired through hierarchical designs to ports that are created at the top level of the instrumented CUV. It is important to note that these ports constitute the interface to the fault injection control unit. The output of circuit instrumentation is a netlist in which fault injection locations are still instantiated as black box.

FPGA Synthesis After circuit instrumentation, the instrumented netlist is passed together with the implementation of fault injection cells to a FPGA vendor specific synthesis tool, e.g., Quartus II for Altera FPGAs. The advantage of this methodology is that different fault model implementations can coexist. In this way, only the vendor specific FPGA synthesis has to be re-invoked in order to change the implementation of fault injection cells. Additionally, the design files of the fault emulator, including the fault injection control unit, the observation / classification unit, the communication interface (UART) as well as the test memory, are fed to the FPGA synthesis tool to complete circuit instrumentation and to connect the respective hardware components. Then, the FPGA synthesis tool synthesizes everything together and configures the FPGA with the resulting bit-stream file.

6.4 Fault Injection Campaign

The emulation-based fault injection campaign used for security verification is implemented as a flow of consecutive executed fault emulations, referred to as fault emulation flow in the following. Additional preparation steps, namely fault generation, test upload and generating a fault-free reference by means of performing a golden run, complete the implementation of the fault injection concept.

A fault emulation flow iterates all fault configurations received over the communication interface, which represent N_e fault experiments executed in a consecutive manner. Since single fault injection times with variable fault durations are configurable, $N_e \leq |\mathcal{F}_{\text{total},n=1,\forall d}|$ different fault configurations can be iterated, where $|\mathcal{F}_{\text{total},n=1,\forall d}| = \binom{|T|}{2} \cdot |\mathcal{F}_{\text{spatial,value}}|$ according to Equation 4.50. Furthermore, $|\mathcal{F}_{\text{spatial,value}}| = |V_u \setminus h|^{|L|} - 1$, where $V_u \setminus h = \{0, 1, b, u\}$. The holding mechanism is realized as a sequence of forcing a logic level and then holding this logic level. Therefore, it is necessary to remove h from V_u for the complexity calculations. In total, $\binom{|T|}{2}$ valid fault durations can be considered, which follows from choosing all possible two-combinations of the set of discrete times T to represent all combinations of the fault injection time t_{inj} and the fault release time t_{release}. The contribution of h to the spatial-value fault injection complexity is therefore covered by the binomial coefficient $\binom{|T|}{2}$ instead. The spatial-value fault injection space $\mathcal{F}_{\text{spatial,value}}$ can be replaced by any of its subspaces.

Note that the optimization that will be presented later in Section 8.4 enables the fault emulation environment to configure multiple fault injection times as well, which allows to configure every fault configuration included in the total fault injection space $\mathcal{F}_{\text{total}}$.

When considering just few different fault durations with low numbers d, the number of performed fault experiments N_e can be approximated by

$$N_e \approx |\mathcal{F}_{\text{spatial,value}}| \cdot N_T \cdot N_D, \tag{6.12}$$

where N_D is the number of considered fault durations and $N_T = |T|$ is the temporal granularity.

Due to the FPGA-based implementation, besides iterating the fault experiments, also the tasks with frequent data exchange between FPGA-board and control software required to configure a particular fault experiment are iterated. These tasks together with the fault experiment constitute a fault emulation, which is discussed next.

A single fault emulation consists of the following four phases.

- Fault configuration upload
- Configuration
- Fault experiment
- Fault classification and result download

These four phases cover all tasks with frequent data exchange between FPGA-board and control software as well as a single fault experiment, during which either a particular bit-flip or stuck-at fault is being injected. Subsequently, an unoptimized setup is presented, during which these phases are executed in a sequence as listed above, which is repeated for each fault emulation. Performance optimization measures minimizing communication overhead, reducing runtime for fault experiments or skipping entire fault emulations are presented later in Chapter 8.

All fault emulation phases performed by the fault emulation environment are detailed subsequently. Since fault emulation phases are repeated N_e times, I give formal descriptions of their runtimes, which are used later in Chapter 8 to define performance benchmarks. After that, the preparation steps are detailed. Test generation itself is out of scope of this work, and therefore, it is assumed that either a functional test or structural test pattern are available for upload. However, in Section 6.5 I detail how a software-based self-test is used to support efficient response observation for processor-based designs.

Fault Configuration Upload Prior to performing a particular fault experiment, the host computer initiates the upload of the respective fault configuration onto the FPGA. Note that without utilizing performance optimization measures the hardware is idling meanwhile, waiting for a new fault configuration. The time in clock cycles that it takes to upload the fault configuration t_{upload} is defined in Equation 6.13.

$$t_{\text{upload}} = (m+2) \cdot \frac{4\,\text{Byte} \cdot f_{\text{FPGA}}}{s_{\text{interface}}} + t_{\text{latency}} \tag{6.13}$$

The configuration upload time t_{upload} depends on the spatial fault multiplicity m, the speed of the communication interface $s_{\text{interface}}$ in $\frac{\text{Byte}}{\text{s}}$ and the clock frequency f_{FPGA} used on the FPGA. The constant 4 Byte is caused by the implementation of configuration commands which require 4 Byte each. The constant 2 is the consequence of considering two temporal properties per fault, namely fault injection time t_{inj} and fault duration d. Note that Equation 6.13 also considers the latency for write access on the communication interface t_{latency}, given in clock cycles. This latency is caused by the operating system of the host computer and prolongs t_{upload} of each fault experiment. Depending on the used operating system and the actual communication interface, t_{latency} increases the upload time t_{upload} drastically. I experienced a latency of about 10 ms in average during my experiments. An optimization that removes this performance bottleneck is presented in Section 8.2.2.

Fault Configuration The fault injection control unit fetches fault configurations and applies these to the next fault experiment. The time in clock cycles that it takes to do this is the configuration time t_{config}, which is defined in Equation 6.14.

$$t_{\text{config}} = m \cdot t_{\text{l}} + t_{\text{t}} + t_{\text{d}} + t_{\text{r}} \tag{6.14}$$

The configuration time t_{config} is composed of the implementation specific times given in clock cycles it takes to configure the fault model and affected fault injection locations ($t_{\text{l}} = 7$), the fault injection time ($t_{\text{t}} = 8$), the fault duration ($t_{\text{d}} = 8$) and to reset the fault mask register ($t_{\text{r}} = 1$).

Fault Experiment The CUV starts processing the test while the configured fault is being injected at the configured time for the configured fault duration. This fault experiment including all relevant timings is illustrated in Figure 6.4. The test duration t_{test} is the time in clock cycles needed by the CUV to process the test. Also the timing for the fault-free reference is depicted, which is determined during the golden run. A timeout t_{timeout} determines when the fault experiment is terminated in case that the CUV does not respond due to the fault injection, e.g. when the device stalls in a state from which it cannot recover.

As illustrated in Figure 6.4, a failure response can be detected at any time between the moment of the fault injection and a configurable timeout. Hence,

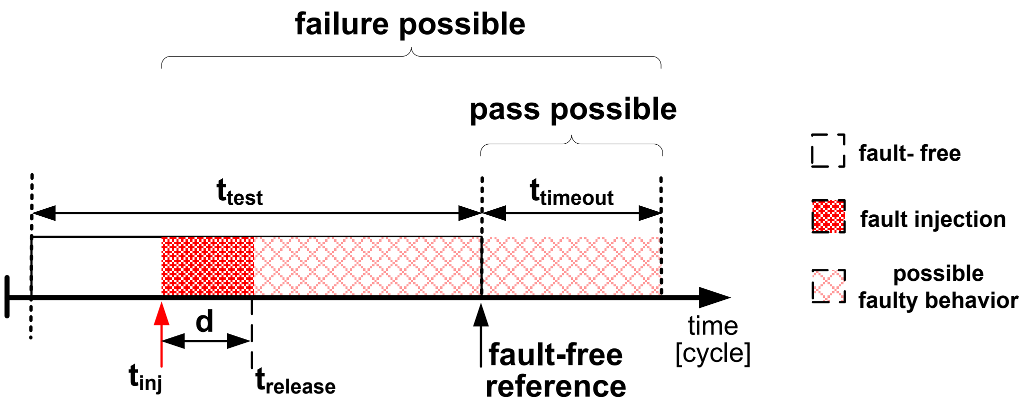


Figure 6.4: Timing diagram for a particular fault experiment.

for a particular fault experiment the actual test duration and also the timeout varies between zero and their maximum value, respectively. Equation 6.15 evaluates the time in clock cycles it takes to perform a single fault experiment.

$$t_{\text{experiment}} \leq t_{\text{test}} + t_{\text{timeout}} \tag{6.15}$$

Note, due to the fault behavior of injected faults, the timing of fault experiments may differ from fault-free fault emulation (fault-induced deviation of timings).

Fault Classification and Result Download The CUV's response is observed and the emulation result is interpreted in hardware for fault classification. This is further discussed in Sections 6.5 and 6.6, where methods for response observation and fault classification are presented in detail. After fault classification, the emulation result is downloaded from the FPGA by the control software. The result download is performed once after completing the fault emulation flow. Note that by taking advantage of the full-duplex communication interface, result download could also be performed in parallel to performing fault experiments.

Next, the preparation steps are outlined. Note that these steps are required to be performed only once before performing fault injection campaigns.

Fault Generation Fault configurations are generated by the control software. The control software provides configuration commands for selecting the fault duration, the fault injection time, the fault model and for configuring affected fault injection locations. Each configuration command requires in total 4 Byte, where 1 Byte is used for a command field and 3 Byte are used for associated data. This is important to note for the performance benchmarks and optimization measures introduced later on.

To perform a fault emulation flow, fault configurations are generated in software covering the desired subset of the total fault injection space[1] $\mathcal{F}_{\text{total}}$. For this, fault selection strategies, such as strategies to select all single faults or specific m-location faults can be applied to select the corresponding subset of the spatial fault injection space $\mathcal{F}_{\text{spatial},n=1}$. By applying the desired fault model type and additionally iterating all temporal fault properties, i.e. all relevant fault injection times $t_{\text{inj}} \in T_i$ and if required different fault durations (single fault injection time with variable fault durations), fault configurations are generated that then represent the selected subset of the total fault injection space $\mathcal{F}_{\text{total},n=1,\forall d}$. Using the optimization that will be presented later in Section 8.4, the fault emulation environment is enabled to configure multiple fault injection times as well, i.e. arbitrary fault configuration included in the total fault injection space $\mathcal{F}_{\text{total}}$ are configurable. To support this feature, instead of iterating all relevant single fault injection times, a set of arbitrary combinations of multiple fault injection times is iterated. A sequence of fault configuration commands implements a particular fault configuration function, hereinafter referred to as fault configuration, which is then uploaded onto the FPGA during the corresponding fault configuration phases of fault experiments and executed as a particular fault experiment.

Test Upload and Golden Run Once the circuit instrumentation and FPGA synthesis is done, the control software uploads a test into a dedicated test memory, from where the test is applied to the CUV during a golden run and also during following fault experiments. The golden run is executed only once in advance to executing fault experiments in order to determine the fault-free reference. This is further detailed in the next section.

6.5 Response Observation

As discussed in Section 5.5, conventional approaches for response observation are unattractive. Approaches that instantiate the CUV twice are unattractive because of limited hardware resources on FPGAs and performing fault-free and faulty emulations in a time-serially fashion constitutes a performance

[1]Without the optimization that will be presented later in Section 8.4 fault configurations are limited to single fault injection times with variable fault durations.

bottleneck. Furthermore, in case of executing functional tests, fault injection may change the program's instruction sequence, which constitutes the test sequence. This causes fault-induced timing deviation in test execution, which may falsify emulation results. Storing results of functional tests in observable memory locations in the memory map of the CUV is also unattractive since it is likely that these are corrupted due to fault propagation.

The methodology for response observation presented subsequently tackles these issues. Moreover, I propose using internal signals as observation points to increase observability. In case that the CUV is a processor design, I present how this concept is supported by software-based self-tests. Complex and often unreliable result evaluation of functional tests is enhanced by means of comparing observation points to at runtime configurable immediate output. For security designs, I propose using a second set of observation points to monitor internal alarm signals of fault countermeasures to enable fault classification for security designs.

The presented methodology is also optimized in several other directions. Neither a second CUV instance is required nor a time-serially execution of golden and faulty emulations is performed. Hence, it is more efficient in terms of both performance and hardware requirements compared to the conventional approaches discussed in Section 5.5. Therefore, the proposed methodology is especially attractive for larger CUVs such as processor-based designs sensitized by functional tests, however, it is not limited to these designs. Furthermore, it allows to configure to what extend fault-induced timing deviation in test execution is tolerated. This is a necessity to prevent false positives during fault classification.

6.5.1 Using Adequate Observation Points

Instead of using only top-level interfaces, internal signals are used as response observation points to check the CUV response in presence of faults. This increases observability and, in turn, also testability.

For security designs deploying fault countermeasures, I propose to split observation points into two categories. The first category indicates whether faults corrupt the CUV's functionality and the second category indicates whether hardware fault countermeasures detect a corrupted functionality:

- **Alarm observation points** for monitoring fault countermeasures
- **Response observation points** for monitoring functional behavior

The combination of these two sets of observation points allows the observation / classification unit of the fault emulator to evaluate whether faults affect the CUV's functionality critically and should be detected by fault countermeasures. This way, fault classification is enabled for security designs.

Alarm Observation Points Usually, fault countermeasures implemented in hardware indicate with internal alarm signals whether a fault has been detected (refer to Section 3.1). Hence, alarm signals are sufficient for reliably observing the effectiveness of fault countermeasures. For instance, an internal alarm signal could be raised when a faulty value has been detected by error detection on the data bus of a microprocessor or when a state machine transitions into an illegal state. For this purpose, the observation / classification unit implements an edge detection for alarm observation points, observing whether an alarm was raised.

Response Observation Points Response observation points are required to detect the incorrect functionality of a CUV in presence of faults. In case of non-processor architectures, these observation points are monitored for appropriate events, e.g. the output of a calculation such as the cypher text of an encryption algorithm which is then compared to the fault-free reference (refer to Section 5.5.2).

6.5.2 Determining the Fault-Free Reference

The expected fault-free reference at response observation points is configured into the observation / classification unit in advance to executing the fault-free golden run, which is initiated by the control software. One observable event is the fault-free reference generated by the CUV at the end of the sensitizing test. Then, the golden run is used to monitor the response observation points to determine the corresponding cycle in which the test ends. This information constitutes the timing of the fault-free reference (depicted in Figure 6.4), which is stored for following fault experiments. A timeout is also configurable, which is realized as offset to the determined timing of the fault-free reference.

6.5.3 Determining Fault Emulation Results

During fault experiments, the observation / classification unit observes the response at observation points and compares it to the previously configured expected fault-free reference. A corrupted circuit behavior, i.e. a failure response, results in a timeout, if the fault-free reference is not detected until the configured timeout. As depicted in Figure 6.4, the fault-free reference can only be detected in the time frame defined by the expected timing of the fault-free reference and the timeout, which is then interpreted as pass. This way, the test result is either pass or timeout, where a timeout corresponds to a failure.

In the next section this concept is improved by introducing a second observable event for a failure response. Since security designs are often realized by processor-based architectures, I present a solution optimized for these circuits.

6.5.4 Observation of Processor Designs

In case of processor architectures, incorrect functionality may exhibit as a corrupted instruction sequence or corrupted data access. Therefore, data and address buses as well as communication interfaces can be used as response observation points. Since utilization of communication interfaces in functional tests constitutes a performance bottleneck, buses should be preferred.

The presented methodology uses the program counter (address bus) of a processor-based CUV to observe its functionality. A test software is used to propagate faults to the program counter. The program counter reflects the addresses of fetched instructions with which the deviation from a correct instruction sequence can be determined. However, the program counter can also be exploited to detect corrupted data access, when the executed functional test is organized as a software-based self-test (SBST). For example, if the functional test double checks the register value versus an immediate value after a write access on it and a jump follows dependent on the test result, then corrupted data is indirectly observable as deviation in the program's instruction sequence. This way, the test software sensitizes a path to the program counter when either the program's instruction sequence or data access is corrupted, i.e. corrupted functionality can be observed at the program counter. In order to prevent that the observation / classification unit is required to compare the entire sequence of the program counter to a fault-free reference, the software-based self-test is written in such a way that only specific values are relevant, each representing an observable event, as detailed next.

In general, software-based self-tests (SBST) are used to sensitize processors when structural testing of processors is technically or economically infeasible (refer to Paragraph Functional Tests in Section 3.4). Software-based self-tests add redundancy in software which allows to check whether an operation fails or passes. There are different ways to add redundancy.

For instance, an operation, e.g. writing a specific value to a register, is performed and then the same register is read back to validate it versus an immediate value. In this way, the functional test is able to check whether the operation passed or failed indicating whether the functional behavior was affected by the fault injection. Another approach is to execute the operation twice and compare the results. The software stores the test result for each operation. At the end of the test, i.e. after testing all operations, the software jumps to either a fail-label or pass-label depending on the test results. These jumps are directly reflected on the program counter. This allows the observation / classification unit of the fault emulator to evaluate the test result in a very generic way by comparing the program counter to two different values, representing the pass- and failure events.

Besides the discussed evaluation of test results, which is the main purpose of the SBST, there is also a trade-off for the following points:

- Reliably evaluating test result (pass, fail)
- High fault coverage
- Short test duration
- High robustness

In order to obtain high fault coverage, the test requires to use all accessible hardware resources in terms of instructions, registers and memory. On the other hand, in order to maintain a good performance, the test needs to be as compact as possible. Basically, this leads to the classic fault coverage problem known from manufacturing testing. Functional tests need to be written in a robust way to prevent that it hangs up too often due to the fault injection. This is important to allow the evaluation of the test result even in case where very active and critical faults are injected.

In general, it is preferable to build tests dedicated to testing specific units and use-cases. The SBSTs used in this thesis makes extensive use of the instruction set, special function registers and perform encryption/decryption to sensitize crypto-processors.

SBST-supported Response Observation In case of applying a functional test to processor-based architectures, the proposed concept for response observation is used to observe the program counter for an address that corresponds to the end of the functional test (pass event) and an additional address that corresponds to a routine that is executed in case of a failure (fail event). Subsequently, these addresses are referred to as pass-label and fail-label, respectively, which correspond to the fault-free reference and the failure response, configured in advance to performing the golden run.

During fault experiments the observation / classification unit observes the program counter and simultaneously compares it to both the previously configured pass-label and fail-label. If neither the configured fault-free response (pass-label) nor the failure response (fail-label) is detected until the configured timeout, then the emulation is terminated. This way, as depicted in Figure 6.4, a failure response can be detected at any time from fault injection until the timeout and a fault-free response (pass-label) can be detected at any time in between the expected timing of the fault-free reference and the timeout. This way, the test result is either pass, fail or timeout, based on which fault classification is performed, as detailed after discussing the advantages of the proposed methodology.

6.5.5 Discussion of Advantages

Subsequently, the advantages of the proposed methodology for response observation are discussed. These include the ability to tolerate timing deviation of fault experiments and the ability to introduce circuit- and test-related events, which increases the applicability of fault emulation, as well as hardware savings and performance gain.

Tolerating Fault-Induced Timing Deviation of Fault Experiments The presented methodology prevents false positives caused by varying timing behavior of fault experiments, which might occur due to the fault injection. For instance, the test could basically pass and the only difference due to the fault injection is that the test execution has been delayed by one cycle. However, in the end the service is provided and also security requirements are not violated. Using the conventional approach that compares the responses of two CUV instances (one faulty instance, one fault-free instance), would cause false positives in emulation results since it is likely that the timing of both emulations differs. As a consequence, these false positives need to be sorted out manually by a test engineer, which is very time-consuming, and hence, needs to be prevented in order to use limited verification times efficiently.

Since very active signals are used as observation points, the conventional approach that expects a single output (refer to Section 5.5.2) cannot be applied either. I addressed this issue by configuring the expected fault-free reference (pass-label), the expected failure response (fail-label) and a configurable timeout. This concept allows to monitor very active signals, while tolerating timing deviation in test execution for which it does not matter whether the test fails or passes. Only in case that the test execution is too corrupted such that the device hangs or the timing is completely off, the timeout is actually reached. This way, the configurable timeout allows to constrain to which extend a timing deviation is tolerated, and hence, helps to prevent false positives in emulation results.

Observing Test-related Events The concept of observing internal signals such as the program counter of a processor can be exploited to create checkers for additional test-related events. A detection of hardware stalls is realized this way, by means of monitoring whether the program counter stalls. This is used to separate timeouts from hardware stalls during fault classification, as further discussed in the next Section 6.6.

Furthermore, this method can be exploited to introduce additional events during fault emulation, which for example is used to define a time frame in which security-relevant operations are executed without manually determining the absolute timing of it. This is realized by extracting the start and end address of a security-critical section of the functional test. These are configured into the observation / classification unit in advance to executing the fault-free golden run, used to determine the corresponding absolute timing as for the pass- and fail-labels. This information is provided in special function registers, enabling the control-software to automatically determine all relevant fault injection times $t_{\text{inj}} \in T_i$ during fault generation. Moreover, time frames of varying length can be excluded from fault injection, such as execution of uncritical software parts that set the design into a certain state of interest for security verification. This way, fault configurations do not have to be updated to take new absolute fault injection times into account after adapting the setup software and, as a result, emulation results of different setups remain comparable. Further, this mechanism is used to constrain fault injection times for a performance optimization discussed in Section 8.2.3.

Hardware Savings and Performance Gain In contrast to existing approaches that uses two CUV instances to generate the fault-free reference and the faulty response in parallel (refer to Section 5.5.1), the golden run needs to be performed only once in advance to performing fault experiments. Since fault injection campaigns execute a huge amount of fault experiments, the additional runtime for a single golden run is negligible. Only a single CUV instance is required for this method, and therefore, the presented method saves about 50% hardware resources on the FPGA. The freed resources enables to apply the fault emulation to larger circuits, such as processor-based security designs, and hence, increases applicability.

Conventional approaches are required to wait for the timeout in all cases except of detecting the fault-free response. This constitutes a considerable performance bottleneck. In contrast, the presented fault emulation environment allows to also configure a failure response that is likely to be generated due to the structure of software-based self-tests. This way, the emulation can already be terminated as soon as either the fault-free response or the failure response is detected. As indicated in Figure 6.4, the failure response can be detected at any time between fault injection and the configurable timeout. This can be explicitly exploited by using a software-based self-test that performs the jump to the fail-label as soon as one of its checks fails. This results in a considerable performance gain. Furthermore, compared to approaches that alternate between golden run and fault experiments, the runtime is considerably decreased by about 50%, i.e. the performance is doubled.

6.6 Fault Classification for Security Designs

The presented methodology for response observation is especially useful to perform fault classification for security designs, for which it is required to decide whether a fault causes a critical CUV behavior. Table 6.1 summarizes all fault classification possibilities and their interpretation within a security context.

A fault causes a critical behavior if the CUV responses with a failure response and the countermeasures of the device do not raise an alarm, and hence, are not detecting the fault. A timeout is interpreted as uncritical within a security context only in case that the device stalls since a stalled device is also not leaking secrets. If a timeout is reached while the processor is not stalling, then it is still executing code. From a security point of view, as long as an alarm has not been raised, their is still a risk for violating security requirements after the defined timeout. Hence, this has to be considered a critical case. All other cases are interpreted as uncritical.

Table 6.1: Fault classification for security designs by interpretation of fault emulation results observed at response observation points and alarm observation points.

response	**alarm**	**no alarm**
failure	uncritical	**critical**
pass	uncritical	uncritical
timeout & stalled	uncritical	uncritical
timeout & not stalled	uncritical	**critical**

6.7 Summary and Discussion

I presented a fault emulation environment used for modeling fault attacks during security verification. The hardware implementation covers single and multiple stuck-at-0/1, set/reset and bit-flip faults, where a single fault injection time and the fault duration are configurable. I used the circuit instrumentation technique, which I completely automated using synthesis-tools while still featuring the ability to control the instrumented design parts. This way, the fault emulation environment supports arbitrary multiple fault injection in combinational and sequential logic. The implemented fault model was formally described for gate level by means of specifying properties according to the the meta fault configuration model presented in Chapter 4.

I implemented fault generation and upload as well as test upload in software to achieve the flexibility required for modeling arbitrary fault attacks. Compared to approaches that generate fault configuration autonomously in hardware, this implementation comes at the cost of performance loss, which motivated me to develop performance optimization measures, presented in Chapter 8.

The presented method for response observation suits processor-based architectures, enabling and optimizing fault classification for security designs. I proposed a very efficient implementation in terms of both performance and hardware requirements. This implementation does not only allow to determine the test result, but also provides the flexibility to define and observe additional events. I used this ability to enable the control software to automatically determine relevant fault injection times during fault generation.

Although the fault emulation environment allows fault injection in combinational cells, I focused on fault injection FFs so far. During my experiments I experienced that circuit instrumentation for combinational logic raises unacceptable issues. Most importantly, because of the huge hardware overhead generated by instrumenting a huge number of combinational cells, its applicability is limited to small circuits. An extensive demand for buffers required to meet setup time constraints, whose availability is limited on FPGAs, often requires to consider only few combinational cells for circuit instrumentation. As a consequence, this renders modeling arbitrary fault attacks in combinational logic impracticable. Furthermore, the clock frequency needs to be reduced in order to meet setup time constraints, resulting in a reduced performance. Since fault emulation performs orders of magnitude better than all other alternatives, I was motivated to develop a concept that still benefits from this high performance and at the same time provides the capability to configure arbitrary multiple faults in combinational logic. These thoughts are picked up and further discussed in detail next in Chapter 7.

Chapter 7

Enhancing Fault Injection in Combinational Logic

This chapter is dedicated to enhancing fault injection in combinational logic and presents an efficient alternative to the conventional stand-alone fault emulation approach presented in Section 6.2.2. I published the idea of the following concepts in [NHS15] and a detailed version of it in [NHHS16].

Next, in Section 7.1, I discuss issues that are encountered when implementing fault injection in combinational logic using FPGA-based fault emulation. This motivates the proposed concept for fault injection in combinational logic, which is presented in Section 7.3 after reviewing conventional approaches briefly in Section 7.2. I extend fault emulation by a software-based pre-processor for fault injection and propagation in combinational logic. For this purpose, combinational and sequential fault propagation is separated. Combinational fault propagation is performed for the fault duration using a software-based method incorporating a SAT-solver in order to determine equivalent faults in sequential logic. Then, the fault emulator presented in the previous chapter is used for further sequential fault propagation by means of injecting the determined equivalent faults into FFs. I outline the implemented fault model in Section 7.3.1, after which I detail the proposed concept in Section 7.3.2. I discuss state equivalence and fault equivalence along with the literature to derive the equivalent relation of transient faults in Section 7.3.3, demonstrating the validity of the presented concept. Implementation details are outlined in Section 7.3.4. Finally, Section 7.4 summarizes and concludes this chapter.

7.1 Motivation

Emulation-based fault modeling demands additional logic for controlling fault injection on the FPGA. This is the case for both fault injection in sequential and combinational logic. However, digital designs usually comprise considerably more combinational cells than sequential cells. Additional logic is inserted into the data path, not only once per register but for each combinational cell. This causes the following issues:

1. The hardware overhead caused by instrumenting a huge amount of combinational cells restricts fault emulation to smaller circuits because of limited hardware resources on the FPGA.

2. The critical timing path gets longer; I experienced a 68% reduced (divided by 3) operating frequency after having instrumented the combinational logic of an 8051-like micro-controller, and Pellegrini et al. [PSC$^+$12] even reported an 80% reduced (divided by 5) operating frequency for a SPARC-V9 architecture.

3. Resources for adding path delays required to fix hold time violations are likely to be exhausted. The reason is the huge number of wires for controlling the fault injection which are routed to combinational cells. These are more likely violating the hold time as the instrumented combinational cell gets closer to the capturing flip-flop.

This renders emulation-based approaches for fault injection in combinational logic unattractive and raises the demand for alternatives.

The method presented subsequently overcomes the discussed issues with respect to combinational fault modeling. I achieve this with a combined approach that utilizes a software-based method to inject faults into combinational cells and to propagate these into sequential cells and then proceeds with fast FPGA-based fault emulation for faults in sequential cells. In this way, the hardware overhead for instrumenting combinational cells is completely removed. As a result and in contrast to existing work, the presented method can be applied to much bigger circuits. Furthermore, the critical timing path of the emulator is not increased, which would be the consequence of instrumenting combinational cells, resulting in a considerable performance improvement, as will be demonstrated in the result section. Hence, it still benefits from the high performance that FPGA-based fault emulation provides, i.e. it is three to five orders of magnitude faster [EMEM14, EVC$^+$09] than simulation and software-based symbolic approaches. Moreover, the presented method is able to inject multiple transient faults into combinational and sequential cells, whereas existing work often focuses on either one or restricts fault injection by instrumenting a subset of combinational cells. Hence, it is perfectly suitable for accelerating complex fault injection campaigns based on layout or structural information.

7.2 Related Work

To limit the hardware overhead, existing emulation-based approaches either instrument only few combinational cells, e.g. [EVC$^+$09, PSC$^+$12] and [KLPB05], or consider fault injection in sequential cells only, e.g. [EMEM14] and [LGPE07]. As a consequence, the former approaches are not able to select arbitrary multiple faults, and hence, these are not suitable as accelerator for extensive multiple fault injection in combinational logic based on selection strategies using e.g. layout information. There is also a second motivation to restricting instrumentation; Pellegrini et al. [PSC$^+$12] reported issues to meet timing constraints when instrumenting random locations in a SPARC-V9 architecture. To evade this problem, they had to partition some of its modules into smaller sub-modules in which they were then able to randomly select few fault locations for circuit instrumentation. That is, they constrained the distribution of fault injection locations in order to meet timing constraints. There are approaches reported in literature that map large designs into multiple FPGAs or on industrial prototyping platforms [DBG$^+$09]. However, the communication between different boards significantly reduces the operational frequency, and hence, the speedup [ESRT15]. Using approaches that only allow fault injection into sequential cells, the vulnerability to faults of the combinational logic cannot be analyzed at all.

There are also simulation- and other software-based approaches available, as reviewed in Section 3.6. Simulation-based implementations that provide fault injection in combinational cells, e.g. [AZS12] and [DNFLR12], are too slow to use limited verification times efficiently. And symbolic software-based approaches, such as presented by Miskov-Zivanov et al. [MZM10] do not scale for industrial circuits, especially when functional tests lasting thousands or ten-thousands of clock cycles or multiple faults are considered.

To increase simulation performance, Aguirre et al. [ABTV07] propose to use fault emulation for sequential fault propagation once fault propagation in combinational logic is simulated. For this, a logic simulator is used to simulate the CUV executing a test until the fault injection time is reached. Built-in simulator commands are used to inject a single fault into combinational logic. After simulating the respective fault injection interval, the errors manifested in sequential logic are captured and mapped onto a fault emulator. Since every simulation starts from the beginning, in average half of simulation time is spent for repeated fault-free simulations. The tool performance could be considerably increased by applying the simulation algorithm proposed by Alexandrescu et al. [AAN02]. Basically Alexandrescu et al. propose to implement a checkpoint for the first time interval. The simulation falls back to this checkpoint after simulating a fault of a given fault set. This is repeated for all faults in this set of faults after which a checkpoint for the next time interval is created for which everything is repeated again. This way, simulation effort is mainly spent for simulating fault effects, but not for repeated simulations of fault-free time intervals. However, Alexandrescu et al. were only interested in analyzing the probability of faults originated in combinational logic to be latched in sequential cells, and therefore, techniques for sequential fault propagation are not considered.

In a security context, it is very likely that physical fault attacks cause multiple faults, as discussed in Section 2.3.2, and has therefore to be considered during security verification. However, when considering multiple fault injection in combinational logic in addition an even larger fault space has to be handled. Hence, fast fault injection environments are required to benefit as much as possible from limited verification times. Unfortunately, fast fault injection environments capable of injecting multiple faults in combinational logic are not available, neither from industry nor academia.

In summary, existing work lacks at least one of the following requirements:

- acceptable performance to benefit from limited verification times,
- arbitrary multiple fault injection to mimic arbitrary physical fault attacks,
- applicability for fault selection strategies based on e.g. layout,
- applicability for industrial designs and long functional tests.

7.3 A Pre-Processor for Combinational Faults

The proposed method is applicable to model single and multiple event transients (SET, MET), which correspond to single and multiple faults in combinational logic. Basically, in the very first cycle of fault propagation, SETs and METs are mapped to equivalent faults at inputs of sequential cells by utilizing Boolean constraint propagation provided by a SAT-solver. Single and multiple event upsets (SEU, MEU), which correspond to single and multiple faults in sequential logic, can be modeled with the used fault emulator by default and without any software-based pre-processing, as presented in Chapter 6.

A cycle accurate transient fault model similar to the one for single transient faults presented in [PHRB11] is used. Fault configuration is limited to one clock cycle and only logical masking is considered. In contrast to [PHRB11], multiple event transients are considered in addition.

The following concept is generalized such that it does not depend on implementation-specific limitations. In order to consider all masking effects, the presented implementation of fault propagation can be replaced by approaches that allow to consider circuit timing. To consider arbitrary fault durations, the presented combinational fault propagation step can be adapted to handle multiple clock cycles of fault propagation. For this it would be required to derive iteratively faulty stimuli for the combinational logic for the fault duration based on the faulty response of the combinational logic.

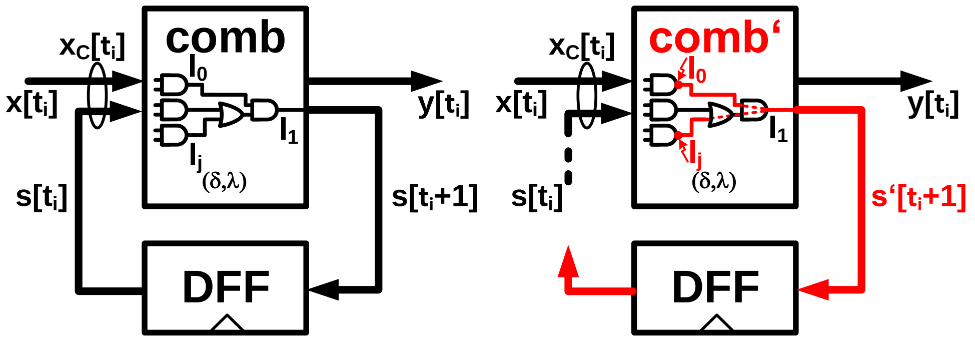
7.3.1 Fault Configuration Model for Combinational Cells

Subsequently, I define the fault configuration model for combinational cells, which is going to be implemented in this chapter. Similarly to the fault configuration model at gate level and the hardware implementation of combinational fault injection cells presented in Sections 6.1.2 and 6.2, multiple and single bit-flip faults are realized. Only single fault injection times with a fault duration of one clock cycle are considered and the hold functionality is therefore not realized. Moreover, stuck-at fault model types are not considered since only one out of two possible values per fault injection location is relevant when a fault duration of exactly one clock cycle is modeled, which is already covered by the bit-flip fault model.

Since only a single time interval $T_i \in T$ is affected, the set of affected time intervals (temporal fault configuration) T_a equals both the fault duration interval T_d and the set of fault injection intervals T_{inj}, so $T_a = T_d = T_{\text{inj}}$, where $T_a, T_d, T_{\text{inj}} \subseteq T$ and $T_{\text{inj}} = [t_{\text{inj}}, t_{\text{release}})$ and $t_{\text{inj}}, t_{\text{release}} \in \mathbb{N}_0$. Furthermore, $d = t_{\text{release}} - t_{\text{inj}} = 1$, where t_i and t_{i+1} of the time interval T_i correspond to t_{inj} and t_{release}, respectively, where $t_i, t_{i+1} \in \mathbb{N}_0$.

The implemented fault model covers the subset $V_{\text{C}} \subset V_u$ of the value configuration space implemented for fault injection in sequential cells, where

$$V_{\text{C}} = \{ b, u \}, \ v \in V_{\text{C}}. \tag{7.1}$$



(a) Fault-free **(b)** Faulty

Figure 7.1: Fault effect in sequential logic caused by a transient fault in combinational logic. (a) Fault-free CUV illustrated as FSM, composed of DFFs and combinational logic. (b) Faulty state transition $s[t_1] \to s'[t_1+1]$ caused by a transient bit-flip fault cBF $= (\varphi'_{\mathrm{BF}}, T_{\mathrm{inj}})$.

Hence, the subset of the total configuration space $\left({V_u}^L\right)^T$ for bit-flip fault configurations in combinational logic, denoted by f_{cBF}, can be simplified to

$$f_{\mathrm{cBF}}(T_i) = \begin{cases} \varphi'_{\mathrm{BF}} & \text{for } T_i = T_{\mathrm{inj}} \\ \varphi_u & \text{otherwise} \end{cases}, \tag{7.2}$$

where $f_{\mathrm{cBF}} \in \left({V_u}^L\right)^T$ defines a family of fault configuration functions that maps a single time interval included in T_{inj} to a particular forcing function $\varphi'_{\mathrm{BF}} \neq \varphi_u$. All other time intervals map to the fault-free forcing function φ_u, where $\varphi'_{\mathrm{BF}}, \varphi_u \in {V_u}^L$. The forcing function φ'_{BF} maps fault injection locations $l_j \in L_{\mathrm{C}}$, where $L_{\mathrm{C}} = \{\, l_0, l_1, \ldots, l_{N_L-1} \,\}$, to values $v \in V_{\mathrm{C}}$. During the fault injection interval T_{inj}, affected fault injection locations $l_j \in L_a$, where $L_a = L_a(\varphi'_{\mathrm{BF}})$, are mapped to $b \in V_{\mathrm{C}}$. Unaffected fault injection locations, so $l_j \notin L_a$, map to the fault-free value $v = u$.

The corresponding description for a parametrized fault, denoted by cBF, is then given by the tuple cBF $= (\varphi'_{\mathrm{BF}}, T_{\mathrm{inj}})$.

7.3.2 Concept

In Figure 7.1a, the state machine (FSM) introduced in Section 6.1.1 is depicted. A faulty state transition of the same FSM is illustrated in Figure 7.1b. Since a cycle accurate fault model is utilized, fault injection into the CUV's combinational logic is possible once per state $s[t_i]$ of the circuit. Therefore, the time index t_i of the state $s[t_i]$ corresponds to a possible fault injection times. In Figure 7.1b, affected fault injection locations $L_a = L_a(\varphi_{\mathrm{BF}}) = \{\, l_0, l_j \,\}$ are depicted as red-colored lightning bolts at the outputs l_0 and l_j of combinational logic gates. Fault propagation through combinational logic is indicated by a dashed red arrow.

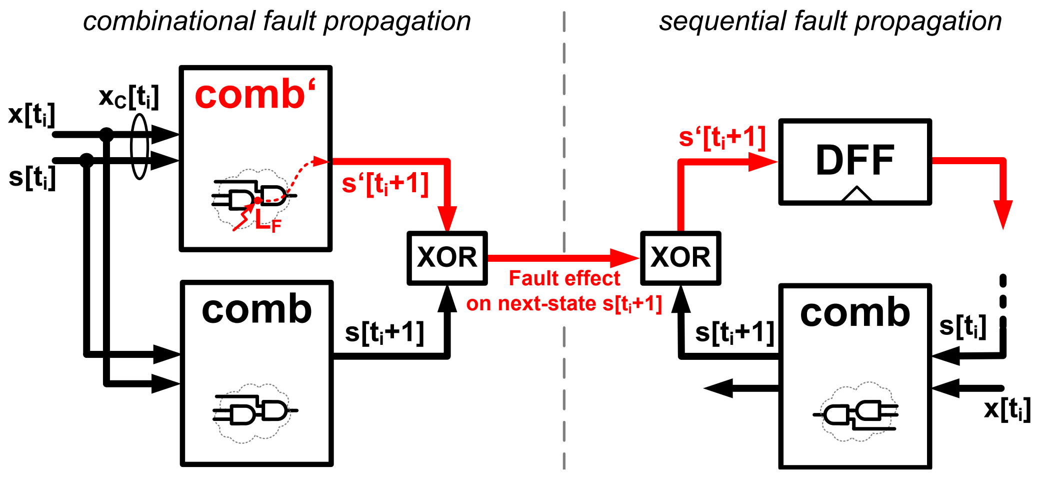
A propagated fault may result in a faulty next-state $s'[t_{i+1}]$ (depicted in red), if latched by DFFs with the next sensitive clock edge. After latching the fault in sequential cells, the fault is present at the corresponding sequential cells' outputs in the next cycle, corrupting the internal state. Hence, the consequence of the transient fault cBF affecting L_a in cycle $t_i = t_{\mathrm{inj}}$ is a faulty state transition $s[t_i] \to s'[t_{i+1}]$. An equivalent fault is now present in sequential cells causing further faulty state transitions, i.e., the fault propagates as an error through the circuit. Note that implementations for a fault duration $d > 1$ and for multiple fault injection times have to consider that the initially injected fault cBF does not only affect the first state transition, but all state transitions during the affected time intervals T_a. When considering timing constraints and electrical masking, fault propagation from the fault site to sequential cells is completed after a respective path delay, which has to be considered in addition.

I propose to split the problem of fault modeling in combinational logic into two steps, as depicted in Figure 7.2. A software-based method is used to inject faults in combinational logic and to propagate these into sequential cells (depicted left). This way, an equivalent sequential fault can be determined (indicated by XOR on the left), which is then mapped onto an FPGA-based fault emulator for further sequential fault (error) propagation (depicted right). This method benefits from both the flexibility of software-based methods and the high performance of emulation techniques.

Subsequently, fault injection in combinational logic, combinational fault propagation, extraction of equivalent faults for fault injection into sequential logic as well as sequential fault propagation are detailed.

Combinational Fault Propagation First (depicted left in Figure 7.2), the assumption[2] function of a SAT-solver is used to model SETs or METs as transient fault cBF in the combinational logic *comb'*. Bit-flips ($v = b$) are injected at affected fault injection locations L_a. The fault injection time is represented by applying the stimulus $x_C[t]$ (composed of top level stimuli $x[t]$ and states $s[t_i]$) at combinational inputs (PPI).

In the very first cycle $t_i = t_{\mathrm{inj}}$ of fault propagation, i.e. the fault injection time, the propagate function of a SAT-solver is used to map cBF to equivalent transient faults at inputs of sequential cells. As depicted in Figure 7.2, equivalent sequential faults are determined by XORing the fault-free next-state



combinational fault propagation
sequential fault propagation
x[t_i]
x_C[t_i]
s[t_i]
comb'
L_F
s'[t_i+1]
XOR
comb
s[t_i+1]
Fault effect on next-state s[t_i+1]
s'[t_i+1]
DFF
XOR
s[t_i+1]
comb
s[t_i]
x[t_i]

Figure 7.2: Separating fault propagation: (left) software-based combinational fault injection and propagation of the fault cBF $= (\varphi'_{\mathrm{BF}}, T_{\mathrm{inj}})$ in state $s[t_i]$. (right) sequential fault propagation, processed by FPGA-based fault emulation, depicted for the first faulty state transition $s[t_i] \rightarrow s'[t_{i+1}]$.

$s[t_{i+1}]$, generated by a fault-free instance of the combinational logic *comb*, and the faulty next-state $s'[t_{i+1}]$, generated by the faulty combinational logic *comb'*.

Sequential Fault Propagation The fault emulator performs fault-free state transitions in advance until the fault injection cycle $t_i = t_{\mathrm{inj}}$ is reached. Then, the determined transient fault at inputs of sequential cells is mapped onto the fault emulator by means of XORing it with the fault-free next-state $s[t_{i+1}]$, which is depicted right in Figure 7.2 for the first faulty state transition $s[t_i] \rightarrow s'[t_{i+1}]$. The result is the faulty next-state $s'[t_{i+1}]$, which is latched by DFFs. That is, an equivalent sequential fault is injected, which is present at outputs of DFFs in the next clock cycle. Now, the fault injection is deactivated and the sequential fault emulation is continued, performing error propagation until the end of the executed test.

7.3.3 Fault Equivalence

Since the proposed method maps faults from combinational logic to equivalent faults in sequential logic, the term fault equivalence needs to be introduced, which goes together with state equivalence and the definitions of

[2]The assumption function can be used to constrain input variables by means of assuming values of Boolean variables.

distinguishable and indistinguishable faults. The following definitions are inline with [KKJ10, BHF97] and were originated for the purpose of optimizing test pattern generation for permanent stuck-at faults in synchronous sequential circuits.

Kohavi et al. define distinguishable states as follows:

Citation 1 (Distinguishable States [KKJ10])**.** Two states, S_i and S_j, of a machine M are distinguishable if and only if there exists at least one finite input sequence that, when applied to M, causes different output sequences depending on whether S_i or S_j is the initial state.

Furthermore, Kohavi et al. define state equivalence as follows:

Citation 2 (Equivalent States [KKJ10])**.** The states S_i and S_j of machine M are said to be equivalent if and only if, for every possible input sequence, the same output sequence is produced regardless of whether S_i or S_j is the initial state. Thus, S_i and S_j are equivalent if there is no input sequence that distinguishes them.

Note, considering advanced testing concepts that observe pseudo primary outputs (PPO), the output sequence also includes the state sequence produced by a state machine. Now based on these considerations, Boppana et al. [BHF97] derive the definition for distinguishable faults.

Citation 3 (Distinguishable Faults [BHF97])**.** A fault pair (F_1, F_2) is said to be distinguishable if there exists an input sequence such that for every pair of initial states S_{F_1} and S_{F_2} of the faulty machines corresponding to faults F_1 and F_2, respectively, the output sequence produced by the faulty machine corresponding to F_1 in response to the input sequence when the machine starts in state S_{F_1} is different from the output sequence produced by the faulty machine corresponding to F_2 in response to the input sequence when the machine starts in state S_{F_2}.[3]

The definition for fault distinguishability says that two faults can be distinguished, when each fault is separately injected into an identical state machine and the corresponding state machines generate different output sequences in response to the same input sequence.

The proposed method maps transient faults from combinational to sequential logic. Based on the discussed definitions, I will derive the relation

[3]The identifier for faults F_1, F_2 (original f_1, f_2) and faulty states S_{F_1}, S_{F_2} (original S_{f_1}, S_{f_2}) are adapted to match the nomenclature of this thesis. In the context of the presented fault configuration model, F_1, F_2 are two arbitrary parametrized faults (see Definition 4.3.8).

of fault equivalence between transient faults in combinational and sequential logic as well as distinguishability and indistinguishability of transient faults. These definitions constitute the base for the proposed method, allowing to map transient faults from combinational to sequential logic. Furthermore, an optimization that collapses the number of faults considered for further fault emulation is derived based on these considerations, which is presented and discussed in Section 8.3.2.

The fault initially injected in combinational logic results in a faulty state, when the fault is not masked, and therefore, is latched into sequential cells. Since the state machine is fault-free until the fault injection time $t_i = t_{\text{inj}}$, the state $s[t_i]$ in which the fault is injected can be considered the initial state, which then in turn may cause a faulty next-state, which results in a faulty sequence of states according to the definition for fault distinguishability given by [BHF97]. As far as fault durations $d > 1$ are considered, the faulty state sequence caused by the faulty state $s'[t_{i+1}]$ is further affected by the injected fault for the remaining fault duration $d - 1$. That is, the fault manifests as additional errors in already faulty states. Note, the propagation delay from the fault site to sequential logic has to be considered in addition, possibly resulting in latching-window masking of fault effects in the last time interval of the fault duration interval, so in time interval $[t_{\text{inj}} + d - 1, t_{\text{release}}) \subseteq T_d$. Note, for the presented implementation $d = 1$. Hence, the last and only time interval of the fault duration interval is the injection interval $T_{\text{inj}} = [t_{\text{inj}}, t_{\text{release}})$, where $t_{\text{release}} = t_{\text{inj}} + 1$.

Based on these considerations, definitions for indistinguishable, distinguishable and equivalent transient faults are given. These definition can be simplified when a single input sequence (particular test) is considered and when the same fault injection time and the same fault duration are considered for the considered faults. The goal is to show that fault equivalence can already be determined as soon as the fault site L_a stops affecting the circuit. Without considering circuit timing, this corresponds to the fault release time t_{release}.

Indistinguishability of Transient Faults Assume the state sequence is observed at pseudo primary outputs in addition to the output sequence. Two transient faults F_1 and F_2 (independent of spatial and temporal properties) injected into identical state machines M_{F_1} and M_{F_2}, respectively, are said to be indistinguishable if the corresponding state machines generate in response to every possible input sequence equivalent state sequences and equivalent output sequences. Now, assuming two transient faults with identical fault injection time and identical fault duration, to determine indistinguishability of the faults under consideration only the fraction of state and output sequences that correspond to the fault duration interval and in addition the propagation delay have to be considered. That is, if the two state machines M_{F_1} and M_{F_2} generate in response to every possible input sequence equivalent state sequences and equivalent output sequences until the moment when the respective faults F_1 and F_2 are released plus the propagation delay, then F_1 and F_2 are said to be indistinguishable. Furthermore, for the state sequence the response of combinational logic is only relevant during the latching-window.

The presented method is applied to perform fault injection campaigns for security verification, where the verified circuit is sensitized by the same input sequence (particular test) during all respective fault experiments. For this purpose, the definition for indistinguishability of transient faults can be relaxed such that indistinguishability is determined only for this test instead of considering all possible input pattern.

Distinguishability of Transient Faults By implication, two faults F_1 and F_2 injected into identical state machines M_{F_1} and M_{F_2}, respectively, are said to be distinguishable if the corresponding state machines generate in response to every possible input sequence nonequivalent state sequences or output sequences. When considering a particular test, the definition for distinguishability of transient faults can be relaxed such that distinguishability is determined only for this particular test.

Transient Fault Equivalence in Combinational and Sequential Logic A transient fault in sequential logic that is equivalent to a transient fault in combinational logic can be determined the moment when indistinguishability can be determined. Therefore, fault equivalence of the initial injected fault in combinational logic and the resulting fault in sequential logic can be determined the moment when state transitions, taking latching-windows into account, are not further affected by the fault site, i.e. the instance of time when fault propagation from the fault site to sequential logic is completed after the fault is released. Note, as far as circuit timing is considered, latching-window masking has to be taken into account in addition.

This can be exemplified with Figure 7.1b. The multiple bit-flip fault at the two locations l_0 and l_j in state $s[t_i]$ is equivalent to a single fault at location l_1, when all inputs of the first stage of AND-gates are sensitized simultaneously by logical '0'. Since l_1 is connected to a PPO, it directly affects the next-state $s[t_{i+1}]$ when it is latched, and hence, an equivalent transient fault in sequential logic can be determined. Note that the used fault emulator injects faults at data inputs of FFs, and therefore, it is sufficient to determine fault equivalence for input pins of sequential cells. This way, the delay of FFs, which would be part of circuit modeling is not a matter for determining transient fault equivalence in combinational and sequential logic.

The implementation that I present in the next section is cycle accurate and fault injection is aligned with the clock. This way, circuit timing and electrical masking has not to be considered, and hence, latching-window masking for DFFs is trivial. That is, in case that a sensitized path to sequential logic exists, faults propagated through this path are not mitigated and, hence, always arrive at inputs of sequential cells. Therefore, $t_{\text{release}} = t_{\text{inj}} + 1$ is the time when fault equivalence is determined. Since fault injection is limited to a duration of exactly one clock cycle, so $d = 1$, faults are only affecting a single state instead of a sequence of possibly already faulty states spanning multiple time intervals. Nevertheless, once a fault manifests as an error in a state, it may propagate sequentially through consecutive states, which I propose to handle with FPGA-based fault emulation.

7.3.4 Implementation

This section presents the tool flow implementing the fault injection environment for transient faults in combinational logic. Implementation details about combinational fault injection and propagation with a SAT-solver is outlined as well. The proposed method suits as a pre-processor for fault emulators that are able to model SEUs and MEUs in sequential cells, as the one presented in Chapter 6.

The software-based pre-processor is implemented using Boolean constraint propagation, provided by the open source SAT-solver MiniSAT [ES]. Boolean constraint propagation of MiniSAT is used instead of fault simulation because fault simulators supporting METs were not available, neither from academia nor industry. Furthermore, fault simulators usually drop a fault from simulation as soon as it is detected at observation points without providing further information such as logic levels of PPOs. The proposed method, however, requires to extract the exact output levels of the combinational logic part for each fault. Propagate functions of SAT-solvers are highly optimized (linear complexity). As long as the circuit input is completely constraint, which is the case in the presented application, conflicts do not occur, and MiniSAT is able to solve the problem with unit clause propagation. Additionally, using multiple instances of MiniSAT, the effort spent is well scalable, where, in contrast to using commercial simulators, license models and license costs can be left out of consideration.

Tool Flow The fault injection environment is depicted in Figure 7.3. The red-colored items highlight the implementation of fault propagation according to Figure 7.2. The tool flow utilizes two EDA tools, a customized version of MiniSAT 2.2.0, an FPGA-based fault emulator and custom scripts written in Perl and TCL.

First, a *preparation step* is performed to generate the input of our methodology. A synthesis tool (Synopsys Design Compiler) divides the CUV into sequential and combinational logic, writes a complete netlist and writes the pure combinational part of the circuit into a second netlist. The synthesis tool is also used to generate a fault list composed of SETs, based on which fault lists composed of METs are generated. Input stimuli for the combinational logic are generated through gate level simulation of the complete netlist. These input stimuli are then used together with the other pure combinational netlist, which is translated into DIMACS[4] CNF[5] format, for combinational fault propagation with MiniSAT. MiniSAT performs a *combinational fault propagation* flow, which is outlined in this section. The result is a list of fault effects captured at inputs of sequential cells, i.e. a sequential fault list, which covers all determined SEUs and MEUs. The sequential fault list can be collapsed to distinguishable faults, referred to as dSEUs and dMEUs, fed to the fault emulator for *sequential fault propagation*. This optimization is detailed in the next Section 8.3.2 together with performance optimizations for fault emulation. Synthesis and logic simulation are automated using TCL scripts. Several scripts, written in Perl, combine all tools and subscripts into a completely automated flow, handle file format conversion and perform result analysis.

Combinational Fault Injection and Propagation The fault injection and propagation flow is implemented using Boolean constraint propagation, provided by the open source SAT-solver MiniSAT [ES]. The source code of MiniSAT was modified to automatically add fault injection capability for each output of combinational cells in its internal CNF representation of the circuit, as depicted for an example AND-gate in Figure 7.4. The original gate is duplicated with an inverted output. An additional multiplexer is used to chose between the original fault-free gate and the duplicated, faulty gate. This can be represented in CNF in a very efficient and easy to generate way.

The same modification is performed for each combinational cell in the circuit. Each gate output is assigned a consecutive number $j \in \{0, \ldots, N_L - 1\}$, where N_L is the number of combinational fault locations $|L|$. Values assigned

[4]Center for Discrete Mathematics and Theoretical Computer Science

[5]Conjunctive Normal Form, excepted by MiniSAT as input

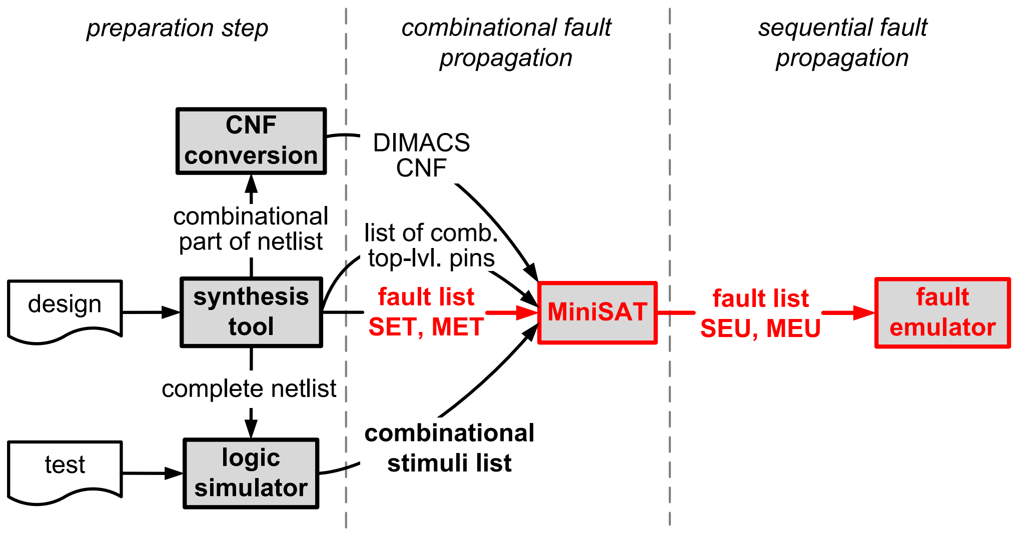


Figure 7.3: Tool flow combining a software-based pre-processor and a fault emulator to enhance fault emulation of faults in combinational logic.

to fault locations are gathered in a vector $(v_1, v_2, \ldots, v_{N_L-1})$ with $v_j \in \{u, b\}$. Each component of this vector defines whether the corresponding gate pin is faulty $(v = b)$ or fault-free $(v = u)$ by applying it to the corresponding multiplexers, using MiniSAT's assumption function. This way, SETs as well as METs can be modeled in a very generic way.

The described approach is effective for single-output gates only. If a gate has multiple outputs, e.g. adders, the gate will be instantiated multiple times. For example, the gate will be instantiated twice for gates with two outputs. For each instance, only one output is connected in the netlist, i.e. the first output of the first instance and the second output of the second instance and so on. Corresponding inputs of all instances are connected in parallel. In this way, the concept of adding fault injection capability to single-output gates is still valid.

The modified version of MiniSAT iterates a combinational fault list composed of configurations for SETs and METs and a list of combinational stimuli, consisting of one stimulus $x_c[t]$ for each fault injection time t_{inj}. A fault-free propagation for each stimulus is performed in advance in order to determine a fault-free reference for all next-states $s[t_{i+1}]$. Then, MiniSAT iterates both the list of combinational stimuli and the fault list in a double nested loop. For each iteration, MiniSAT determines the fault effect on the next-state $(s[t_{i+1}]\ xor\ s'[t_{i+1}])$ according to Figure 7.2, resulting in a sequential fault list composed of configurations for SEUs and MEUs. Two lists of CNF variables specify the appropriate top-level inputs and outputs where stimuli are applied and the result is captured, respectively. As already mentioned

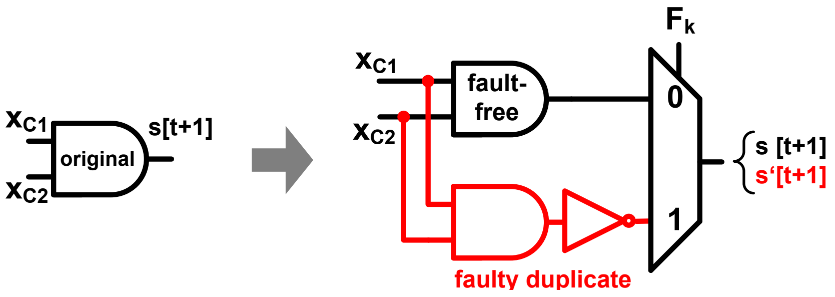


Figure 7.4: Adding fault injection capability to an AND-gate.

and further discussed in Section 8.3.2, the resulting fault list is optimized by collapsing it to distinguishable representatives before it is then used as input for sequential fault emulation. In order to increase the tool performance, multiple MiniSAT instances are executed in parallel for independent subsets of the combinational fault list.

Sequential Fault Propagation The fault emulator presented in Chapter 6 and illustrated in Figure 6.1 is utilized for sequential fault propagation.

7.4 Summary and Discussion

In this chapter I proposed to use a software-based method to mimic transient faults in combinational logic (SETs and METs). Boolean constraint propagation provided by the SAT solver MiniSAT was used to inject transient faults and to propagate these through combinational logic into sequential cells. The implementation is based on the gate level fault configuration model introduced in Section 6.1, which was restrained for fault injection in combinational logic and matches the conventional stand-alone emulator-based implementation presented in Section 6.2.2. This way, comparison to conventional existing work is enabled when discussing the results.

The proposed software-based method is able to determine transient faults in sequential cells that are equivalent to the initially injected transient faults in combinational logic. This way, a software-based pre-processor for the FPGA-based fault emulator presented in Chapter 6 was developed that enables to benefit from the high performance fault emulation provides without instrumenting combinational logic for fault injection. I demonstrated the validity of the proposed method by means of deriving the equivalence relation of transient faults from definitions for state equivalence and fault indistinguishability given in literature.

As I am going to show with the results in Chapter 9, I successfully removed the discussed disadvantages of emulation-based fault injection in combinational logic, increasing its applicability for larger circuits. Contrarily to conventional approaches, hardware overhead and timing constraints are not an issue anymore, and therefore, fault injection in combinational logic is not restricted to few locations selected at random. As a result, the presented method allows to perform fault injection campaigns using selection strategies e.g. based on layout, and hence, suits mimicking physical fault attacks. Next in Chapter 8, I am going to propose performance optimizations which make the proposed method also faster than conventional stand-alone emulation-based fault injection in combinational logic.

A cycle accurate fault model is used, which in general poses a more pessimistic approach compared to fault models with finer temporal granularity. It is assumed that transient faults are caused synchronously with the clock and for a duration that matches the clock period, such that electrical masking has not to be considered and latching-window masking is trivial. However, using cycle accurate fault models the computational effort is drastically reduced, which enables exhaustive fault injection. Furthermore, cycle accurate fault models allow to focus on cases where faults actually lead to errors and are already applicable early in the development cycle when low-level information such as timings are not available [PHRB11]. This is of importance for validating fault countermeasures [PHRB11] and for identifying security flaws early during circuit design [PTH$^+$15].

The concepts are generalized to allow refined implementations that may enable the configuration of fault durations and may consider more detailed circuit models that consider timings and all masking effects. For example, to consider a fault duration where multiple consecutive clock cycles are affected by a fault, for each affected clock cycle one iteration of SAT-solving can be implemented. For this, the fault effect at outputs of the combinational logic together with the fault-free stimulus of unaffected inputs needs to be considered as stimulus for fault injection in the combinational logic in the next affected clock cycle. Recent advances in SAT-solving enable to consider timing models in Boolean functions [SBP15], which could pose a promising alternative to simulation-based approaches in order to apply fault models that consider a more precise time granularity for fault durations (shorter than a clock cycle). Alternatively, it is possible to cover fault effects of low-level fault models including electrical and latching-window masking as well with the proposed method by means of modeling setup and hold time violations for an acceptable amount of affected FFs. For this, all possible combinations of multiple faults in the affected FFs determined by the SAT-based method could be mapped onto the fault emulator to cover these effects.

Chapter 8

Performance Optimizations and a Feature for Multiple Fault Injection Times

I presented the performance optimization measures that aim on fighting the communication bottleneck in [NHN$^+$14] and in [NHRS15]. I published performance optimization measures that aim on skipping equivalent fault experiments when applying the software-based pre-processing for enhancing fault injection in combinational logic in [NHHS16]. I discussed performance optimization measures that aim on reducing the emulation runtime using a silent fault optimization in [NR11].

My goal in this thesis is to build a fault emulation environment that can be used to model arbitrary fault attacks, for which complex fault injection campaigns need to be performed requiring to iterate huge sets of fault configurations. So far, I presented concepts and respective implementations capable of injecting arbitrary multiple faults with a configurable single fault injection time and configurable fault durations (the implementation of fault injection in combinational logic is limited to fault duration $d = 1$). That is, the proposed concepts support the total fault injection space $\mathcal{F}_{\text{total}}$ completely with the exception of multiple fault injection times.

The supported configurability comes at the cost of performance loss because of data exchange between the controlling software components and the FPGA-based fault emulator. The goal of this chapter is to close the gap between speed and configurability of fault emulation environments. For this purpose, I introduce performance optimization measures, which allow to reach the optimal performance, so far only provided by autonomous approaches, while supporting all fault configurations included in the total fault injection space $\mathcal{F}_{\text{total}}$ at runtime.

Next, performance benchmarks are introduced, which are used in Chapter 9 to determine the performance gain provided by the proposed performance optimizations. Performance optimization measures presented in Section 8.2 aim on fighting the communication bottleneck between software components and FPGA, for which I provide experimental results in Section 9.2. In Section 8.3, performance optimization measures are presented that focus on shortening fault experiments and skipping equivalent fault experiments. The effectiveness of one of these measures is demonstrated and its limits are discussed in Section 9.3.

Finally, a feature for enabling multiple fault injection times as generic as possible is proposed. This feature allows to configure all temporal fault properties including an arbitrary number of fault injection times. It therefore enables to configure any arbitrary fault configuration included in the total fault injection space $\mathcal{F}_{\text{total}}$ without limitations.

8.1 Performance Benchmarks

The time in clock cycles it takes to execute an entire fault emulation flow, denoted by t_{flow}, is evaluated by

$$t_{\text{flow}} = N_e \cdot (t_{\text{upload}} + t_{\text{config}}) + \sum_{i=1}^{N_e} t_{\text{experiment}_i}. \tag{8.1}$$

N_e, t_{upload}, t_{config} and $t_{\text{experiment}}$ were introduced earlier in Section 6.4. Note that t_{upload} and t_{config} depend on the spatial fault multiplicity m of injected faults, which is assumed to be constant for a fault emulation flow. Moreover, $t_{\text{experiment}}$ varies dependent on the fault effects of injected faults, i.e. dependent on whether and when the respective faults manifest as internal error or failure (external error). The average runtime for one fault emulation, denoted by t_{run} and given in clock cycles, is a benchmark for the performance, which is evaluated by Equation 8.2. The theoretical optimal performance is reached if the average runtime t_{run} equals the test duration t_{test}, which is the time needed by the CUV to process the sensitizing test, as introduced earlier in Section 6.4.

$$t_{\text{run}} = \frac{t_{\text{flow}}}{N_e} \tag{8.2}$$

With increasing spatial fault multiplicity m more data needs to be uploaded and configured onto the FPGA-based fault emulator. Therefore, t_{upload} and t_{config}, and as a consequence also the average runtime t_{run}, increase with m. Hence, the performance of a fault emulation flow gets worse the more fault injection locations are considered to be affected when modeling multiple faults.

I define the workload of the hardware as a second performance benchmark given in percent. Equation 8.3 defines the workload as the fraction of the time in which the fault emulation environment is actually performing fault experiments, i.e. the net execution time for fault experiments excluding communication and configuration overhead. It is the ratio of the time it takes to perform all fault experiments to the time it takes to perform an entire fault emulation flow including communication and configuration overhead. So,

$$\begin{aligned} \mathit{workload} &= \left(\frac{1}{t_{\text{flow}}} \cdot \sum_{i=1}^{N_e} t_{\text{experiment}_i} \right) \cdot 100 \\ &= \left(1 - N_e \cdot \frac{t_{\text{idle}} + t_{\text{config}}}{t_{\text{flow}}} \right) \cdot 100, \end{aligned} \tag{8.3}$$

where for an unoptimized emulation environment $t_{\text{idle}} = t_{\text{upload}}$. As shown by Equation 8.3, either the configuration time t_{config} or the idle time t_{idle} has to be minimized in order to maximize the workload. Therefore, Section 8.2 presents three different measures for minimizing idle times and configuration times in hardware resulting in a performance improvement.

8.2 Fighting the Communication Bottleneck

Performance optimization measures presented subsequently aim on removing the communication bottleneck between software components and hardware components of the fault emulation. In contrast to autonomous approaches that handle fault generation autonomously in hardware like the one presented in e.g. [LGPE05, LGPE07], I propose to generate faults in software and to upload these onto the FPGA. This way, the required ability to configure arbitrary faults at runtime is provided. However, I utilize three different optimization measures to reach the theoretical optimal performance that was so far only reached by autonomous approaches. Note that just increasing the speed of the communication interface would not be sufficient enough to reach this high performance. Idle times in between consecutively executed fault experiments and latencies caused by the operating system of the host computer executing the control software would still prevent reaching such a performance. But in fact, the presented measures would benefit from a faster communication interface.

8.2.1 Configuration Data Overhead Reduction

Data overhead reduction is realized by optimizing the communication protocol of the controlling software components and the FPGA. For this purpose, configuration commands are optimized and only the delta between fault configurations of two consecutive fault experiments is configured.

Optimized Configuration Commands A sequence of configuration commands is transmitted by the control software to the fault injection control unit, which is synthesized onto the FPGA. These configure the fault properties fault model type, affected fault injection locations, fault injection time and the fault duration. A configuration sequence is finalized by a run command, which starts the fault experiment.

The selection of the fault model type and configuring the affected fault injection locations is combined together in a single command, which has to be issued individually for each affected fault injection location. This command also has the ability to reset the fault mask register in advance because the old value is not reset by default. This allows incremental configuration of multiple faults, where configurations of previous fault experiments can be reused to reduce communication overhead, as detailed in the next paragraph. This concept is also applied to the other configuration commands. Furthermore, the ability to automatically start the fault experiment with every configuration command prevents calling an additional start command per fault experiment. Note that this optimization does not limit configurability at all. The way the configuration protocol and its commands are constructed reduces communication overhead drastically, and therefore, constitutes a performance optimization.

Delta Configuration Additionally, I reduce the data overhead of the communication by incremental fault configuration where only the delta between fault configurations of two consecutive fault experiments is configured. Thus, less data has to be considered for fault configurations, which reduces the upload time and the configuration time. In the best case only a single configuration command needs to be issued to reconfigure e.g. a temporal property or to configure an additional affected fault injection location and automatically starts execution of the fault experiment, while reusing all the other fault properties previously configured for preceding fault experiments. Since the configuration of multiple affected fault injection locations generates the highest amount of data, the configuration of the spatial fault property in the fault mask register needs to be changed as seldom as possible when minimizing configuration data, to maximize the performance. In order to achieve this, I utilize loop structures during fault generation performed by the control software so that for each multiple fault (outer loop) different fault durations (inner loop) and fault injection times (most inner loop) are applied.

A new definition for the optimized average upload time t_{upload} per fault configuration of a fault emulation flow that iterates N_D fault durations and N_T single fault injection times is given by Equation 8.4. Note that a constant spatial fault multiplicity m is assumed in Equation 8.4, i.e. the optimized average upload time t_{upload} is given for the m-th subset of the spatial-value fault injection space $\mathcal{F}_{\text{spatial,value},m}$. The respective average configuration time t_{config} is given by Equation 8.5. Refer to Equation 6.13 and 6.14 for the definitions that apply to the unoptimized fault emulation presented in Chapter 6. The new definitions now consider the utilized loop-structures, where the terms $\frac{m}{N_T \cdot N_D}$, $\frac{1}{N_T}$ and 1 correspond to the impact of the outer loop, the impact of the inner loop and the impact of the most inner loop on the performance, respectively. The more different fault injection times N_T and fault durations N_D are considered for a fault injection campaign, implemented as fault emulation flow, the more t_{upload} and t_{config} decrease. This results in a considerable performance improvement compared to the unoptimized fault emulation, as further discussed in the result chapter (Chapter 9). The definitions for the constants $t_{\text{l}} = 7$, $t_{\text{t}} = 8$, $t_{\text{d}} = 8$ and $t_{\text{r}} = 1$ defined in Section 6.4 still apply to this measure.

$$t_{\text{upload}} = \left(\frac{m}{N_T \cdot N_D} + \frac{1}{N_T} + 1 \right) \cdot \frac{4\ \mathit{Bytes} \cdot f}{s_{\text{interface}}} + t_{\text{latency}} \tag{8.4}$$

$$t_{\text{config}} = \frac{t_{\text{l}} \cdot m + t_{\text{r}}}{N_T \cdot N_D} + \frac{t_{\text{d}}}{N_T} + t_{\text{t}} \tag{8.5}$$

The optimizations presented in this section were based on reducing the generated configuration data, where configurability is not limited at all. However, although the data uploaded onto the FPGA is minimized there are still idle times in which the hardware is waiting for new configurations. Note that increasing the speed of the communication interface would also reduce idle times in hardware, however, a faster communication interface cannot eliminate idle times. The next section aims on eliminating idle times completely.

8.2.2 Parallelizing Fault Experiments and Configuration Upload

The second measure focuses on reducing idle times in hardware. I add the ability to perform fault experiments in parallel to uploading a stream of fault configurations. For this purpose, a FIFO is utilized in hardware to buffer fault configurations. Once the first fault configuration is uploaded onto the hardware, the fault injection control unit fetches fault configurations from the FIFO until a run command is fetched and applies these for the next fault experiment. The fault experiment is then executed according to Figure 6.4, after which the fault configurations for the next fault experiment are fetched immediately. In parallel to this, the control software is continuously uploading further configurations over the communication interface, while it also monitors the fill level of the FIFO. The upload is only interrupted if the buffer is full or no more fault configurations are available. The control software starts uploading fault configurations again when the fill level of the buffer drops below a set value. Since a full duplex communication interface is used, the same concept can be applied to the download of emulation results.

This measure optimizes fault emulation flows in two different ways. First, the parallelization eliminates idle times ($t_{\text{idle}} = 0$) in between fault experiments if the average experiment time is longer than or equal to the average upload time ($t_{\text{experiment}} \geq t_{\text{upload}}$). If the average experiment time is shorter than the upload time ($t_{\text{experiment}} < t_{\text{upload}}$), then the idle time is still decreased, however, it is then evaluated by $t_{\text{idle}} = t_{\text{upload}} - t_{\text{experiment}}$. Minimization of idle states maximizes the workload of the hardware according to Equation 8.3 and also shortens the runtime t_{run}. Second, uploading a stream of fault configurations minimizes write accesses on the communication interface. Therefore, latency for write accesses caused by the operating system of the host computer influences the upload time t_{upload} and the performance only once per fault emulation flow instead of once per fault configuration. I combined this measure with the measures for data overhead reduction described in Section 8.2.1 and measured the resulting performance. The results are discussed in Section 9.2.

8.2.3 Sub-Selection of Fault Injection Cells

This measure comes as a trade-off between performance and configurability. It adds the ability to select a subset L_{sub} of all fault injection cells (locations) L, i.e. $L_{\text{sub}} \subset L$. Depending on the implementation, a specific number of fault injection locations $|L_{\text{sub}}| = K$ can be selected, where $K = 32$ in the following description. Only fault injection cells included in L_{sub} are considered for configuring the spatial property of faults. This allows to reduce the required configuration data when configuring multiple faults. Because of the reduction in configuration data, this fault selection technique is faster than state-of-the-art fault masking approaches. However, it comes at the cost of limiting configurability to faults with spatial fault multiplicity $m \leq K$ since $K = 32$ fault injection cells are 'sub-selected'. That is, only faults with a spatial fault multiplicity in the limits $m = 1$ to $m = 32$ can be configured.

Figure 8.1 illustrates an example for configuring a two-location fault utilizing this measure. From left to right is depicted: a subset of all fault injection cells L_{sub}, a subset mask and the instrumented CUV containing $N_L = |L|$ fault injection cells. In the depicted example, the subset of fault injection cells L_{sub} selects the first 32 fault injection cells, so $L_{\text{sub}} = \{ l_0, \ldots, l_{30}, l_{31} \}$. A subset mask including $K = 32$ bits, which are addressed by the index $k \in \{0, ..., K-1\}$, determines which of these selected fault injection cells are affected and are hence faulty during a particular fault experiment. If a bit is set to '1' in the subset mask, then the corresponding entry in the subset of fault injection cells L_{sub} is enabled (*f_en='1'*). As depicted by the red colored entries in the example in Figure 8.1, the entries of L_{sub} with index $k = 0$ and $k = 1$ are enabled by the subset mask. That is, the subset mask selects l_0 and l_1 to be affected. The fault injection cells $\{ l_{32}, \ldots, l_{N_L-2}, l_{N_L-1} \}$ (dashed gray rectangles) are not selected and remain fault-free by default (*f_en='0'*). This way, a 2-location fault with spatial fault property $L_a = \{ l_0, l_1 \}$ and spatial fault multiplicity $m = |L_a| = 2$ is configured, where $L_a \subseteq L_{\text{sub}} \subset L$.

This concept is used to configure the fault mask register depicted earlier in Figure 6.1, which is able to select any arbitrary multiple fault covered by $\mathcal{F}_{\text{spatial}}$, for which $|\mathcal{F}_{\text{spatial}}| = 2^{N_L} - 1$ possibilities exist (refer to Equation 4.23). However, when using the sub-selection of fault injection cells, single to K-location faults are configurable by configuring both the appropriate subset of fault injection cells L_{sub} and the subset mask. The number of different subset masks, denoted by n_{masks}, that can be applied without re-configuring the entries included in L_{sub} can be anywhere in the limits $0 < n_{\text{masks}} < 2^K$, which covers every spatial fault property included in the power set $\mathcal{P}(L_{\text{sub}})$. This is, for each configuration of L_{sub} one out of $|\mathcal{P}(L_{\text{sub}})| = 2^K - 1$ faults can be configured. Hence, according to Equation 4.21, the spatial fault injection space is limited to $\bigcup_{m=1}^{K} \mathcal{F}_{\text{spatial},m}$, where $K = 32$.

The subset of fault injection cells as well as the subset mask are configurable at runtime without re-synthesizing the design. Note that the entries of L_{sub} are binary coded (24 bit each). They can point to any fault injection cell and do not have to be put in order.

The data reduction, and hence, the performance of this measure increase the more subset masks are applied without re-configuring L_{sub}, i.e. with increasing n_{masks} per L_a. When configuring faults with spatial fault multiplicity $m \geq 2$ the data overhead for configuring a new L_{sub} is already eliminated when configuring 32 different multiple faults without re-configuring L_{sub}, so when $n_{\text{masks}} \geq 32$. In fact, configuring a new multiple fault for an already configured L_{sub} is then more efficient as configuring a single fault when using the other two measures for optimizing the performance presented in the previous Sections 8.2.1 and 8.2.2.

Figure 8.1 is a simplified illustration. In the actual implementation, multiple configurations of L_{sub} are stored in a RAM. Multiple configurations of subset masks are stored in a second RAM. The fault injection control unit fetches configurations from these RAMs in order to configure fault experiments. The RAM contents are uploaded preliminary to executing the first fault experiment, representing an entire fault emulation flow. Similar to autonomous approaches, like the one discussed in [LGPE05], this measure loops over all possible fault injection times in hardware, resulting in a further data reduction. However, the first and the last fault injection time to be considered in this loop can be configured at runtime using the mechanism presented and discussed earlier in Section 6.5.5 (Observing Test-related Events). Hence, a set of considered fault injection times can be chosen individually to focus on a relevant time frame of the executed test such as encryption and decryption of a crypto-accelerator. The fault duration is configured as described in Section 6.4.

Note that this measure is not combined with the measure for parallelizing fault experiments and configuration upload, presented in Section 8.2.2. In comparison to the measure presented in Section 8.2.1, the data overhead is further reduced since less data is generated for configuring spatial as well as temporal fault properties.

8.2.4 Summary

In this section, I presented performance optimizations that minimize data overhead and eliminate idle times in hardware. Next, in Chapter 9, I am going to discuss the performance results of these measures. I am going to demonstrate that fault emulation reaches the optimal performance when applying these optimizations, i.e. the performance of fault-free emulations is reached, even if faults with a spatial fault multiplicity in the hundreds are configured. Note that I observed such a high spatial fault multiplicity rarely when propagating multiple faults from combinational logic into sequential cells using the software-based pre-processing proposed in Chapter 7 for enhancing combinational fault injection. In order to further optimize performance, the emulation runtime can be reduced, on which I focus next.

8.3 Reducing Emulation Runtime

Basically, there are two different ways to reduce the fault emulation runtime:

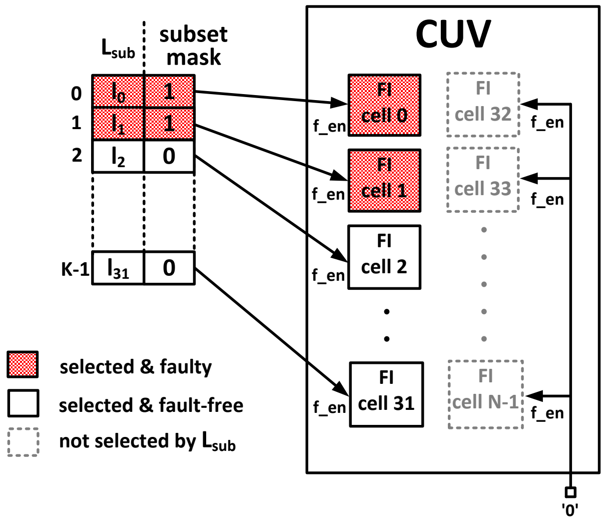


Figure 8.1: Performance optimization measure based on sub-selection of fault injection cells. A subset L_{sub} including K fault injection cells is selected in which the spatial fault property $L_a \subseteq L_{\text{sub}}$ of multiple faults can be configured by applying a subset mask (depicted for a 2-location fault).

- Shorten the execution time for fault experiments by determining emulation results earlier than it takes to execute the test in the fault-free case, i.e. $t_{\text{experiment}} < t_{\text{test}}$.
- Skipping equivalent fault experiments entirely, i.e. $t_{\text{experiment}} = 0$.

Concepts for both cases are detailed in the next two subsections.

8.3.1 Shorten Fault Experiments

Subsequently, I present two concepts that allow to shorten the execution time of fault experiments.

Optimized Software-based Self-test When sensitizing the circuit under verification (CUV) with a software-based self-test (SBST), the test can be built such that faults are propagated as soon as possible to observation points. Instead of storing the test result of a tested operation and performing the final jump to either the pass-label or fail-label at the end of the test, a jump to the fail-label can be performed as soon as the first operation fails. Note that this optimization does not introduce any communication or hardware overhead. Furthermore, it is more effective if less faults are detected by fault countermeasures. Therefore, this optimization helps to reduce turnaround times during the development phase of the CUV when countermeasures are missing or when countermeasures still require improvement.

Silent Fault Optimization The next optimization needs to be implemented in hardware. The idea is to terminate the emulation as soon as a fault disappears from the CUV due to masking and overwriting effects, which is also referred to as silent fault as discussed in Section 3.5.2. For this, it is necessary to compare the entire state of the faulty CUV to a golden reference in a bit-wise fashion. In order to achieve a performance gain, the comparison needs to be performed in hardware at runtime, and therefore, a second CUV instance is required as golden reference. Such an optimization was already proposed by Lopéz-Ongil et al. [LGPE05]. However, Lopéz-Ongil et al. did not consider processor-based architectures. These are especially challenging since faults can propagate into memory. In this case, the entire memory content of the faulty and the fault-free memory needs to be compared at runtime [NR11]. For this, the data of each memory line needs to be fetched to be compared, which would take at least a clock cycle per line during which the emulation would be required to be halted. The emulation time would be prolonged by a factor that equals the number of compared memory lines. In order to avoid this, write operations of both CUV instances to the respective memories can be observed at the memory interfaces.

Neelesh Halinge, a student who I supervised during his Master's thesis [Hal14], implemented a concept in hardware on the FPGA that is able to determine at runtime whether and which memory locations are affected by faults. Basically, a flag per memory line is stored in the block RAM on the FPGA, identifying corrupted memory locations. Unfortunately, this memory as well as the CUV memories need to be flushed in advance to performing the next fault experiment to invalidate the stored data. This would again require to access every memory location, however, now once per fault experiment instead of once per clock cycle. To prevent this, an additional run count incremented for every fault experiment is stored per memory line, which indicates whether its data and the associated valid flag was stored in the actual or in a previous run. This way, only if the run count overflows the flags and the CUV memory needs to be reset.

By adjusting the width of the run counter a trade-off between performance and hardware overhead can be controlled. When using, e.g., a 16 bit counter, every 2^{16}-th fault experiment the run counter overflows, and therefore, the memories need to be reset. Furthermore, dual-ported RAMs and a twice as high clock frequency than the CUV clock was used to increase performance.

With this setup it takes only $\frac{1}{4}$ clock cycle in average per fault experiment to reset the 2^{16} memories lines and associated valid flags of an 8051-like microcontroller, which is negligible. An additional counter is used to trace the number of corrupted memory lines.

In summary, this optimization proved to be effective also for processor-based architectures. The execution time for fault experiments is reduced in average such that $t_{\text{experiment}} < t_{\text{test}}$. However, it comes with a considerable hardware overhead due to duplicating the CUV, adding huge comparators and adding additional memory blocks. Note that this is acceptable only in case that there are enough hardware resources available on the FPGA. For further implementation details and detailed performance results as well as the hardware requirements, I refer to the Master's thesis of Neelesh Halinge [Hal14].

8.3.2 Skipping Equivalent Fault Experiments

The performance optimizations discussed subsequently focus on skipping equivalent fault experiments entirely.

Latent Fault Optimization I propose the concept of an optimization that is able to determine equivalent fault experiments in case that a fault is temporarily latent for a time frame and because of this the next consecutive fault experiment would configure an equivalent fault. For this to work, the fault properties in between consecutive fault experiments are required to be identical except of the fault injection time, which has to be incremented. So, $t_{\text{inj}} = t_i$, where i is incremented from zero to $t_{\text{test}} - 1$ in between fault experiments. Then at least one of the other fault properties, namely fault duration, fault model type and spatial properties is changed and the fault injection time is reset to t_0 and incremented again in between consecutive fault experiments. This is how fault injection campaigns can be configured when using the delta configuration optimization presented earlier in Section 8.2.1.

The latent fault optimization basically traces the fault propagation for fault experiments. Based on this, the clock cycle in which the fault is propagating the first time, denoted by t_p, is determined. As long as $t_{\text{inj}} < t_p$ of following fault experiments, the fault injection would result in an equivalent transient fault since the respective circuit's state sequences of both fault experiments with fault injection times $t_i < t_p$ and $t_i + 1 < t_p$ are equivalent (refer to Section 7.3.3). The latent fault optimization measure detects such equivalent fault experiments and skips these entirely. For fault experiments with $t_i \geq t_p$ the trace of fault propagation is updated to determine the next t_p.

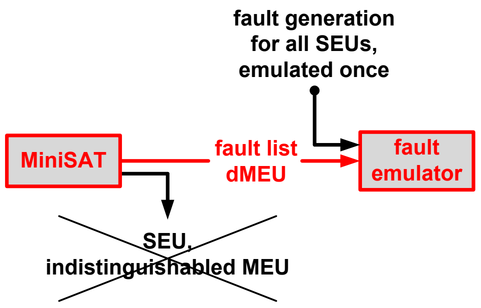

fault generation for all SEUs, emulated once
MiniSAT
fault list dMEU
fault emulator
SEU, indistinguishabled MEU

Figure 8.2: Performance optimization for modeling SETs and METs based on selecting distinguishable faults (dMEUs) from MiniSAT's result for sequential fault propagation and emulating all SEUs once in advance.

The implementation of the latent fault optimization is built on the silent fault optimization measure, which allows to trace fault propagation into memory of processor-based architectures. To determine fault propagation, the state difference of the faulty CUV and the golden reference is stored, representing the state difference of the previous cycle. An additional comparator determines the difference of the state difference of the previous cycle and the current cycle, which is zero when the fault has not propagated. For implementation details and performance results as well as the hardware requirements, I refer to the Master's thesis of Neelesh Halinge [Hal14].

Using this optimization, about 70% of fault experiments when injecting single faults are skipped entirely, which constitutes a good performance optimization. As for the silent optimization, the hardware overhead is considerable. However, when already using the silent fault optimization, the latent fault optimization adds an acceptable hardware overhead.

Emulating Distinguishable Faults This optimization measure reduces the amount of faults mapped onto the fault emulator when utilizing the software-based pre-processing proposed in Chapter 7 for enhancing combinational fault injection. The concept is depicted in Figure 8.2.

The idea is to only map distinguishable faults in sequential logic onto the fault emulator. That is, only faults whose temporal properties (fault injection time) and spatial properties (affected sequential cells) differ from any other determined fault (refer to Section 7.3.3) are considered for further fault emulation. Furthermore, I propose to emulate all single faults in sequential cells (SEUs) for all possible fault injection times once in advance to performing fault injection campaigns for multiple faults in combinational logic (METs).

This way, all SETs and METs that result in SEUs are already covered for subsequent fault injection campaigns, and hence, only distinguishable MEUs (dMEUs) need to be further considered for sequential fault propagation using the fault emulator. Thus, all SEUs and indistinguishable MEUs determined with MiniSAT can be removed from the fault list that is fed to the fault emulator, as indicated in Figure 8.2. These optimizations were derived from the following fault propagation results.

Among other experiments, which are further detailed in Chapter 9, I performed a fault injection campaign during which 22.4 million SETs are injected into an 8051-like microcontroller. In total, only 46.9% of these SETs are propagated and latched by sequential logic (37.6% SEUs and 9.3% MEUs). Only 3.8% of analyzed SETs result in distinguishable MEUs (dMEUs). When considering only dMEUs for further fault emulation on the FPGA, this provides a considerable optimization. I performed similar fault injection campaigns for 2-location METs, 3-location METs and 10-location METs, generated at random. When increasing the spatial fault multiplicity of METs, it is more likely that MEUs are latched, while the number of SEUs decreases and distinguishable faults in sequential logic are generated less frequently. I noticed that equivalent MEUs with fault multiplicities in the tens and rarely in the hundreds are latched. Because of this, it is important that the fault emulator handles single as well as multiple faults in sequential cells with fault multiplicities in the hundreds without performance loss compared to fault-free test runs. As discussed earlier and will be shown in the result chapter, I achieved this with the previously presented optimizations for data overhead reduction (Section 8.2.1) and parallelizing fault experiments and configuration upload (Section 8.2.2).

8.4 Multiple Fault Injection Times

The implementation of the fault emulation environment presented in Chapter 6 is limited to single fault injection times. In this section, I present a feature that enables to configure arbitrary multiple fault injection times, for which I propose an efficient implementation.

The obvious solution would be to add multiple configuration registers to store the considered fault injection times. For instance, five registers would allow to configure five different fault injection times. The problem here is that for each fault injection time also the other fault properties need to be configured such as the affected fault injection locations. Otherwise, the fault properties cannot be varied for different fault injection times. However, implementing the fault mask register multiple times is not a viable option due to the hardware overhead that would be generated. Also, requesting the required configurations from software is not a viable solution since it would reintroduce a communication bottleneck, not only once per fault experiment, but once per configured fault injection time. That is, this implementation would not scale for multiple fault injection times.

Therefore, I propose a different solution that is enabled by the hardware buffers for configuration commands, arranged as FIFO. The idea is to stop the emulation briefly when the next fault injection time is reached to fetch the associated configurations from the FIFO. For this purpose, I introduced an additional configuration command, pause@time, which is configured during the configuration phase and causes the emulator to pause at a configured time. Whenever the emulator pauses, it starts fetching commands from the FIFO again until a run command is fetched. This way, any fault property associated to the respective fault injection time can be reconfigured. Furthermore, when also reconfiguring pause@time during the re-configuration phases, arbitrary multiple fault injection times can be configured.

This feature now enables fault emulation in general and the presented fault emulator to model any arbitrary fault configuration defined in the total fault injection space $\mathcal{F}_{\text{total}}$ without any limitations. Furthermore, the chosen implementation is efficient since simply an additional register for storing the time to pause the emulation and a comparator to determine whether the configured pause time is reached is required to be implemented.

8.5 Summary and Discussion

I introduced several performance optimization measures that aimed on reducing the communication bottleneck, shorten fault experiments and skipping equivalent fault experiments. As I am going to demonstrate next in Chapter 9, these optimizations enable an emulation performance that was by now only reached by approaches that generate fault configurations autonomously on the FPGA. Autonomous fault generation on the FPGA limits however configurability, which renders these approaches impracticable for mimicking fault attacks.

Maintaining the ability to configure arbitrary faults is mandatory in the security context to allow the selection of faults that can be caused by physical fault attacks. Applying the presented performance optimization measures to the presented FPGA-based fault emulator, the optimal performance is reached. Moreover, this way unlimited configurability is provided at the same time, making it suitable for mimicking arbitrary fault attacks in the security context.

Implementing buffers for fault configurations on the FPGA and introducing the additional configuration command for pausing the emulation at a configurable time allows to support fault configurations with multiple fault injection times. Thus, this feature enables fault emulation to model any fault configuration defined in the total fault injection space $\mathcal{F}_{\text{total}}$ without limitations. As long as the buffer does not run out of configurations, the performance is only reduced by an additional configuration time t_{config} per fault injection time. This is acceptable since usually it is only required to configure a few fault injection times and the introduced time overhead is small compared to the test duration t_{test} of realistic functional tests.

In summary, besides closing the gap between speed and configurability of multiple fault emulation environments, I presented the first FPGA-based fault emulation environment that supports arbitrary fault configuration including multiple fault injection times.

Chapter 9

Experimental Results

I published the results I am going to present in Section 9.2 in [NHN$^+$14] and in [NHRS15]. Results for the software-based pre-processing, proposed to enhance fault injection in combinational logic, were published in [NHHS16].

In this chapter I am going to discuss the results of experimental evaluation of the proposed emulation techniques, including the proposed software-based pre-processing enhancing fault injection in combinational logic as well as selected performance optimization measures. I demonstrate with the results that the proposed techniques increase applicability and performance, while the required configurability for emulating arbitrary fault attacks is also supported. This includes fault attacks on sequential and combinational logic in complex designs such as processor-based security designs, which are usually sensitized by long functional tests. For this purpose it is important to evaluate how the performance develops dependent on varying test durations and also dependent on varying fault multiplicities.

Next, in Section 9.1 I outline the circuits under verification (CUV) for which I performed fault injection campaigns. Then, in Section 9.2 I focus on fault injection in sequential logic (SEUs and MEUs). I demonstrate the effectiveness of the performance optimization measures proposed in Section 8.2 to fight the communication bottleneck of fault emulation environments. These enable to maximize configurability without performance loss compared to fault-free emulations. This is required to benefit from limited verification times as much as possible when mimicking arbitrary fault attacks.

After this, in Section 9.3, I present results of fault injection campaigns that inject single and multiple faults in combinational logic (SETs and METs) as well as single faults in sequential logic (SEUs). I present fault propagation results to discuss the effectiveness of the performance optimizations proposed in Section 8.3.2 to skip equivalent fault experiments based on determining distinguishable faults in sequential logic (dMEUs). Finally, in Section 9.3.1 I demonstrate the applicability of the method proposed in Chapter 7 for fault injection in combinational logic, discuss its performance and provide comparison to existing work. Compared to existing work, the proposed technique is applicable to larger circuits and provides a considerably better performance.

9.1 Verified Security Controllers

I applied the presented fault emulation, including the software-based preprocessing for enhancing fault injection in combinational logic and the FPGA-based fault emulation, to two different microcontroller designs used for security applications, namely *80251-mc* and *8051-mc*. These are the circuits under verification (CUV) for which I demonstrate applicability, configurability and performance. For fault injection in sequential cells, all FFs are replaced by fault injection cells during circuit instrumentation. The respective setups are detailed next. Changes to these setups for fault injection in combinational logic are detailed in Section 9.3.

80251 Microcontroller Circuit *80251-mc* is an 80251-like test design with a hardware accelerator for AES (Advanced Encryption Standard) and triple DES (Data Encryption Standard), provided by Infineon Technologies AG for this study. Circuit *80251-mc* is composed of $N_{\mathrm{FF}} = 8051$ FFs and $N_{\mathrm{C}} = 56,641$ combinational cells. This circuit is an industrial design, which implements state-of-the-art fault countermeasures, for which I demonstrate the industrial applicability of proposed fault emulation techniques.

To sensitize the relevant circuit parts, either one of three different functional tests, namely *cpu*, *aes* and *des*, with test durations t_{test} of 79,000, 6800 and 5553 clock cycles, respectively, were executed by circuit *80251-mc* during fault injection campaigns.

8051 Microcontroller The second circuit is an 8051-like microcontroller (*8051-mc*), which is smaller than circuit *80251-mc*. Circuit *8051-mc* is based on the IP core of Oregano Systems [Ore02]. The control unit, the ALU and parts of the data paths of the processor core are protected by fault countermeasures based on Hemming codes, parities and component duplication, which have been implemented by Stephan Janssen as part of his Diploma thesis [Jan09]. Circuit *8051-mc* is composed of $N_{\mathrm{FF}} = 1911$ FFs and $N_{\mathrm{C}} = 10,029$ combinational cells. This circuit is used for performance benchmarks and comparison to existing work.

Since not all components were protected by fault countermeasures, a significant proportion of emulated faults were identified as critical faults during my experiments. Therefore, circuit *8051-mc* was suitable for sanity checks and for evaluating the effectiveness of the proposed fault emulation techniques during their development.

I built a set of tests with test durations t_{test} of 256, 1000, 2000, 3000 and 4000 clock cycles, which is used to benchmark the performance of fault injection in sequential logic. Two tests, namely test *isax1* and *isax2*, lasting 11,851 and 23,811 clock cycles, respectively, are used to benchmark the performance of fault injection in combinational logic. These tests cover the instruction set architecture and are chosen to evaluate the impact of the test duration on the performance. Either one of these tests sensitizes circuit *8051-mc* during fault injection campaigns.

9.2 Fault Injection in Sequential Logic

In this section I discuss results for fault injection in sequential logic of circuit *8051-mc* using the FPGA-based fault emulator presented in Chapter 6. First, the hardware requirements are discussed. Then, I am going to present performance results to demonstrate that optimal performance is achieved with the proposed fault emulation techniques while the configurability required for mimicking arbitrary fault attacks is provided. Focus lies on evaluating the effectiveness of the three performance optimization measures proposed in Section 8.2 to fight the communication bottle neck when increasing the spatial fault multiplicity.

Hardware Requirements Table 9.1 outlines the amount of logic cells needed on the FPGA for implementing three different setups for circuit *8051-mc*. The first row (*CUV*) lists the hardware required by the CUV including the UART used as communication interface, but without any fault injection capability and without the emulator's hardware components. The UART is included since it is required to setup the CUV including test upload. This setup is used as the reference for calculating the hardware overhead. The hardware overhead is listed in column Δ *LEs* (logic elements).

The second row *CUV, measure 1+2* lists the hardware requirements for the setup that includes the instrumented CUV and the hardware of the fault emulation environment. The first and the second performance optimization measures - data overhead reduction (Section 8.2.1) as well as parallelizing fault experiments and configuration upload (Section 8.2.2) - are applied in this setup. Compared to the reference in the first row, this setup results in a hardware overhead of 134%. This is reasonable and comparable to other state-of-the-art implementations since introducing a fault mask register already doubles the required FFs. Further hardware overhead is caused by fault injection cells and for implementing the fault injection control unit and observation / classification unit, which manage the communication and control the fault experiments. In addition, several special function registers are implemented, which are assessable from the control software to monitor the progress of the fault emulator.

The implementation of the third performance optimization measure - sub-selection of fault injection cells (Section 8.2.3) - is built on the implementation of the first and second optimization measures. Therefore, the third row *CUV, measure 1+2+3* lists the total hardware requirements when implementing all three performance optimization measures proposed to fight the communication bottleneck. The implementation of the third measure causes an additional hardware overhead of 26% resulting in a total of 160% hardware overhead. Note that using this setup for modeling fault attacks during security verification, the verification engineer can choose at runtime without re-synthesizing the design to use either the combination of the first and the second optimization measures or the third performance optimization measure. In addition to the hardware overhead listed in Table 9.1, block-RAM on the FPGA is used for implementing communication buffers and realizing the sub-selection of fault injection cells. The amount of block RAM can be adapted pre-synthesis in HDL.

Table 9.1: Hardware requirements of the fault emulation environment when applying performance optimization measures fighting the communication bottleneck: flip flops (FFs), combinational functions (CFs), logic elements (LEs) and hardware overhead (Δ LEs).

	FFs	**CFs**	**LEs**	**Δ LEs [%]**
CUV	2053	6575	7172	0
CUV, measure 1+2	4887	16,400	16,806	134
CUV, measure 1+2+3	5731	18,300	18,653	160

Performance of Measures 1 and 2 I measured the performance in terms of runtime and workload when applying the first two optimization measures. For each measurement point, a fault injection campaign consisting of consecutive fault experiments is performed. The spatial fault multiplicity m is constant per flow and is plotted on the x-axes of every result plot. During a fault injection campaign, spatial fault properties are configured for $N_T = 255$ fault injection times and for a single fault duration of one clock cycle, i.e. $d = 1$ and $N_D = 1$. I chose this configuration to keep the measurements

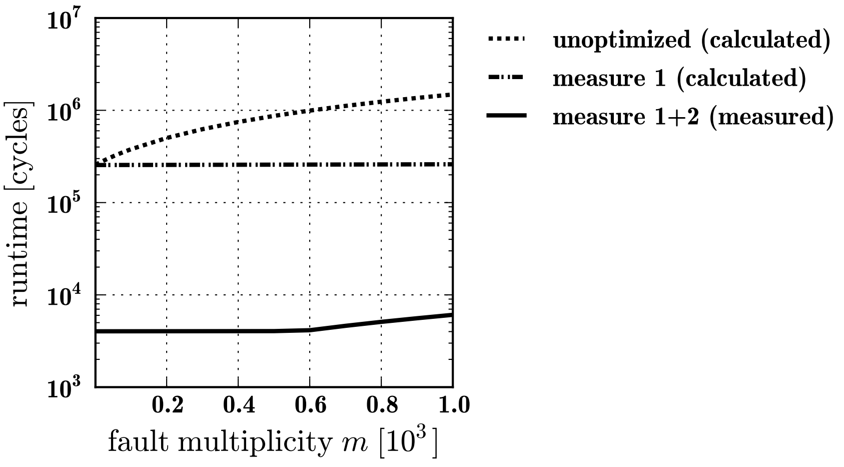
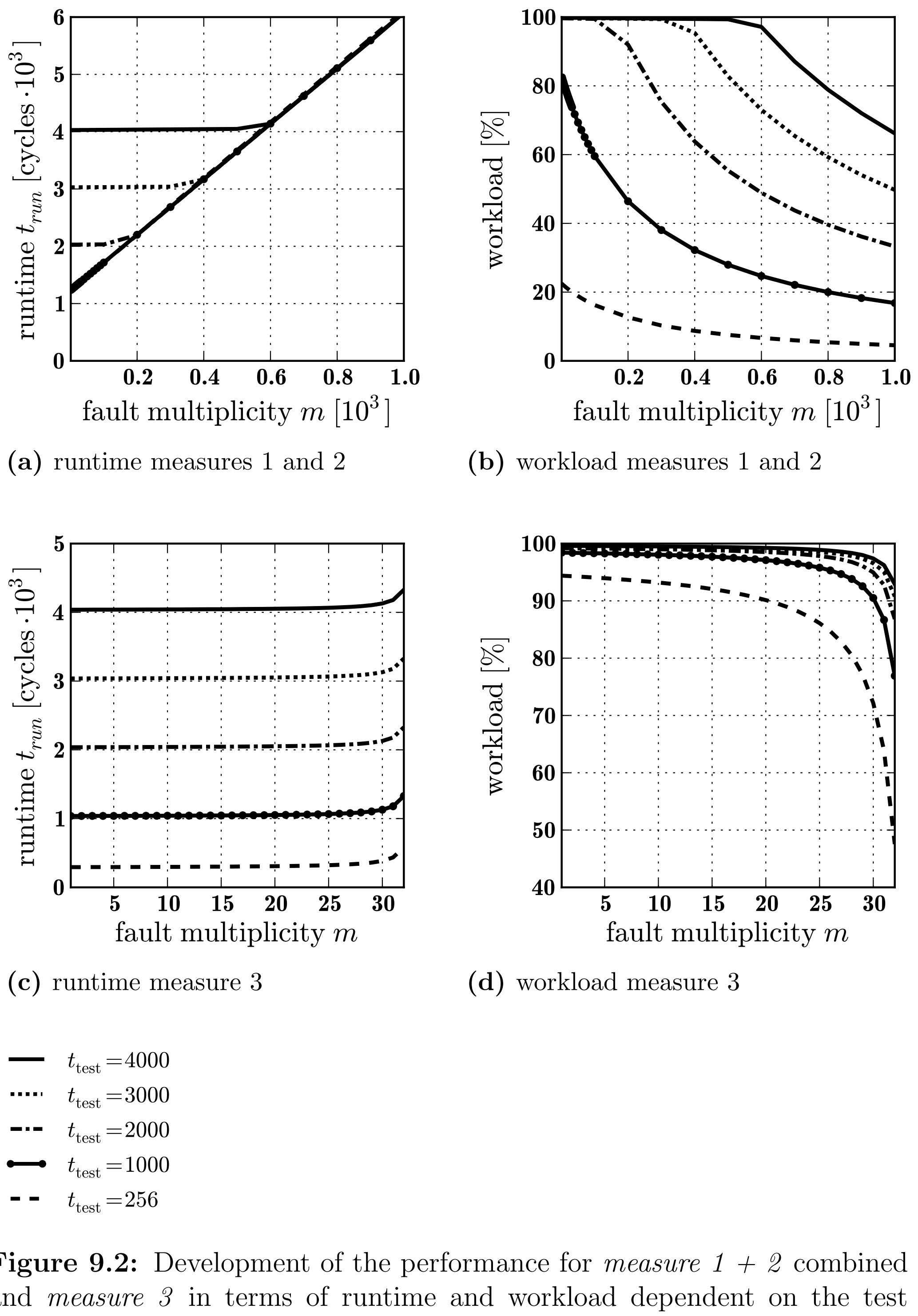


Figure 9.1: Development of the performance dependent on the spatial fault multiplicity for three different setups for a test duration $t_{test} = 4000$: unoptimized, measures 1 and measure 1 + 2 combined.

reasonable and comparable to state-of-the-art approaches that do not support configurable fault durations. Note that according to Equation 8.5, all optimization measures (also measure 3) would perform better if higher values would have been chosen for N_T and N_D. Time measurements are in clock cycles and start when the first fault configuration command is received by the fault injection control unit.

In Figure 9.1 the runtime t_{run} of three different setups for a test duration t_{test} of 4000 clock cycles with increasing spatial fault multiplicity is depicted. Figure 9.1 illustrates how the runtime depends on the spatial fault multiplicity and how the runtime is improved when applying the first two performance optimization measures. The first curve illustrates the performance of an unoptimized setup (*unoptimized*), calculated according to Equations 6.13, 6.14 and 8.2. The second curve shows the performance when the first measure (*measure 1*) - data overhead reduction - is applied, calculated according to Equations 8.2, 8.4 and 8.5. For these calculations the UART is setup with 912.384 Baud, which corresponds to $s_{\text{interface}} = 81\,\frac{\text{KByte}}{\text{s}}$ using one start bit and two stop bits. The average write access latency on the UART caused by the operating system is about 10 ms, which corresponds to $t_{\text{latency}} = 250.000$ cycle at a clock frequency $f_{\text{FPGA}} = 25\,\text{MHz}$. The third curve illustrates the performance, measured within the FPGA, when combining the first and the second measure (*measures 1+2*) - data overhead reduction as well as parallelizing fault experiments and configuration upload. Note that a logarithmic scale is used on the y-axes. The runtime for the unoptimized



(a) runtime measures 1 and 2

(b) workload measures 1 and 2

(c) runtime measure 3

(d) workload measure 3

Figure 9.2: Development of the performance for *measure 1 + 2* combined and *measure 3* in terms of runtime and workload dependent on the test duration t_{test} and the spatial fault multiplicity m.

setup is dominated by the communication interface, resulting in a linear curve with a high slope. As more fault injection locations are affected in a fault experiment, i.e. with increasing spatial fault multiplicity m, the runtime increases, which results in a decreased performance. By applying the data overhead reduction (*measure 1*), the slope is decreased drastically resulting in a performance gain. Note that both curves (*unoptimized* and *measure 1*) are actually linear and have a slope that is greater than zero. When the second measure (*measure 1+2*) - parallelizing fault experiments and configuration upload - is additionally applied, then the runtime is decreased by two orders of magnitude. The performance gain is achieved due to uploading a stream of configurations and introducing buffers in hardware. This minimizes idle times in hardware and write access latency caused by the operating system gets negligible since it influences the performance only once per fault emulation flow instead of once per fault configuration command. Furthermore, the runtime is constant until faults with a spatial fault multiplicity of $m \geq 600$ are modeled. This is the break point where the communication interface limits the performance again, causing idle times in hardware. For the executed tests the communication bottleneck is completely eliminated if faults with $m \leq 600$ are injected. Until this break point is reached, the fault emulation environment performs as good as the theoretical optimum, i.e. the runtime equals the test duration of fault-free emulations, so $t_{\text{run}} = t_{\text{test}} = 4000$. So far, this performance was only supposed to be reached by autonomous approaches, which are not suitable for modeling fault attacks because of lacking configurability. Contrarily, the presented fault emulation environment does not limit configurability at all and performs as good as the fastest state-of-the-art environments for a wide range of multiple faults. Note that a spatial fault multiplicity of $m = 600$ is high enough to model fault attacks that affect hundreds of cells at once without performance loss. Usually security verification focus on faults with one to two orders of magnitude lower spatial fault multiplicities. Furthermore, I noticed during the experiments that target on fault injection in combinational logic, which I am going to detail later, that even multiple faults in combinational logic (MET) with a multiplicity $m = 10$ rarely cause multiple faults in sequential logic (MEU) with multiplicities in the hundreds. Note that all configurations are supported at runtime without re-synthesizing the design. Hence, the configurability is comparable to simulation-based approaches, which closes the gap between performance and configurability of FPGA-based fault emulation.

Figure 9.2a illustrates the impact of the test duration t_{test} on the runtime, where t_{test} is varied between 255 and 4000 clock cycles. Note that now a linear scale is used on the y-axes. The curve for $t_{\text{test}} = 4000$ in Figure 9.2a and the curve (*measure 1+2*) in Figure 9.1 are the exact same curves but with different scales on the y-axes. The curves for the two shortest tests $t_{\text{test}} = 256$ and $t_{\text{test}} = 1000$ overlap in Figure 9.2a. For these tests, the communication interface dominates the performance because it takes more time to transmit the configurations than the hardware needs to process the test. Therefore, the runtime only depends on the spatial fault multiplicity, and it is independent of the test duration. For tests longer than 1200 clock cycles, determined by the intersection between the curve $t_{\text{test}} = 1000$ and the y-axes, there is a range of fault multiplicities for which the runtime is constant and almost equals the test duration, which is the theoretical optimum. The range of multiple faults for which a constant performance can be expected gets wider as the test duration gets longer. Hence, the optimization measures 1 and 2 perform better as the test gets longer. Note, for setups for which the performance does not reach the theoretical optimum the impact of the communication interface on the performance is nevertheless drastically decreased over the entire range of all possible multiple faults (compared to the unoptimized setup in Figure 9.1), even if high fault multiplicities and short tests are considered. Moreover, note that I only performed rather short tests for benchmarking the runtime. For tests longer than approximately $t_{\text{test}} = 15000$ cycles, the runtime is constant, reaching the theoretical optimum over the entire range of all possible multiple faults ($m \leq N_L$).

Figure 9.2b illustrates how the workload of the hardware depends on the test duration. The workload of the hardware gets better for longer tests and is close to 100% when the runtime in Figure 9.2a is constant. For very short tests ($t_{\text{test}} = 256$) the workload is not better than 23% even for single faults.

Performance of Measure 3 The third performance optimization measure - sub-selection of fault injection cells - provides a further performance optimization, especially for short tests. The runtime and the workload when applying only this measure are illustrated in Figure 9.2c and Figure 9.2d, respectively. Since the presented implementation is limited to $m \leq 32$, the x-axis is limited to 32. As shown, the runtimes are close to the actual test durations used in the measurements, even for the shortest test ($t_{\text{test}} = 256$). Compared to Figure 9.2b, the workload of the hardware is drastically increased starting at 94% for the shortest test. Note that I measured a worst case scenario in which a very low number of subset masks $n_{\text{masks}} \leq 32$ is utilized. The increasing slope, especially for $m \geq 30$, is the consequence of this. Note that in the best case scenario $n_{\text{masks}} = 2^K - 1 = 2^{32} - 1$ for the discussed implementation.

9.3 Fault Injection in Combinational and Sequential Logic

In this section I am going to discuss results for fault injection campaigns during which single and multiple faults were injected into combinational logic (SETs and METs) using the software-based pre-processing for enhancing fault injection in combinational logic, which was proposed in Chapter 7. Additionally, single faults are injected into sequential logic (SEUs) in advance, as proposed in Section 8.3.2 to maximize the performance of the presented techniques. I performed fault injection in both circuits, namely *8051-mc* and *80251-mc*, with varying fault multiplicities and varying tests. In all following experiments, faults were injected with a duration of exactly $d = 1$ clock cycle, i.e. the number of considered fault durations is $N_D = 1$. Moreover, the performance optimization measures 1 and 2 - data overhead reduction (Section 8.2.1) as well as parallelizing fault experiments and configuration upload (Section 8.2.2) - are applied. The respective setups of fault injection campaigns including the purpose of sensitizing tests, iterated fault sets and considered fault injection times are detailed in the corresponding sections.

9.3.1 Combinational Fault Propagation Results

In this section I evaluated how faults propagate from combinational logic into sequential logic to determine the fraction of faults that result in SEUs and MEUs, which are then collapsed to respective subsets including only distinguishable SEUs (dSEUs) and distinguishable MEUs (dMEUs). Along with the presented fault propagation results, I show the benefit and limits of the performance optimizations that aim on reducing the amount of faults mapped onto the fault emulator by means of considering only distinguishable faults, as proposed in Section 8.3.2. These optimizations are based on skipping equivalent fault emulations by means of considering only distinguishable faults in sequential logic for further sequential fault propagation after combinational fault propagation.

Table 9.2 outlines the number of faults caused by SETs and METs in combinational logic that are latched by sequential cells, exemplary for circuit *8051-mc*, which was sensitized by test *isax1*. These results constitute a summary of fault lists consisting of faults in sequential cells that are equivalent to the injected faults in combinational logic, which were determined for further sequential fault propagation using fault emulation. In the first row of Table 9.2 the results of a fault injection campaign are outlined, during which 22.40 million SETs (spatial fault multiplicity $m = 1$) were being injected in combinational logic during 22.40 million individual fault experiments. As listed in column *total*, 46.9% of injected SETs were propagated and latched by sequential logic, where 37.6% of injected SETs were latched as equivalent SEUs in sequential cells and 9.3% of injected SETs were latched as equivalent MEUs in sequential cells. 53.1% of all injected SETs were not latched, and thus, were masked logically. Only 21% of the injected SETs led to distinguishable faults in sequential cells, including both dSEUs and dMEUs. If SEUs are injected in advance, as proposed in Section 8.3.2, only dMEUs need to be considered for further fault emulation on the FPGA. This provides a very good optimization for SETs since only 3.8% of injected SETs resulted in dMEUs, and hence, further fault emulation can be skipped for 96.2% of performed fault experiments. In summary, considering only SEUs and dMEUs during sequential fault propagation when performing fault injection campaigns for SETs and METs reduces the effort spent during fault emulation drastically, which constitutes a considerable optimization.

In order to analyze the impact on the performance when increasing fault multiplicity, I performed three different fault injection campaigns for METs. METs were generated at random with a constant fault multiplicity m, where $m = 2$ for the first, $m = 3$ for the second and $m = 10$ for the third fault injection campaign. With increasing spatial fault multiplicity of METs, it was more likely that MEUs were latched, while the number of latched SEUs decreased drastically. Furthermore, distinguishable faults in sequential logic, so both dSEUs and dMEUs, were generated less frequently with increasing spatial fault multiplicity of METs. As a consequence, the optimization potential is only effective for lower fault multiplicities (< 10). For example, in the fault injection campaign during which METs with spatial fault multiplicity $m = 10$ were injected, almost all injected METs were latched by sequential cells (99.8%), which branch into 1.4% latched SEUs and 98.3% latched MEUs.

Note that METs were generated at random only for benchmark purposes and it is likely that several different circuit functions are affected with higher multiplicities of randomly generated METs. Contrarily, when using layout or structural information to model for example laser fault attacks as presented in [PTH+15, PHB+14, VML+14], METs would be injected in a local area with a high probability to affect the same circuit functionality. METs with a high locality considerably limit fault propagation into other circuit areas, which, in turn, generates less pessimistic results w.r.t. latched MEUs and dMEUs. Note that precise attacks with high locality of faults are assumed to be more powerful [PHB+14, PTH+15]. That is, the proposed optimizations are nevertheless effective for fault injection campaigns that mimic, e.g., precise laser fault attacks.

Table 9.2: Faults in combinational cells latched by sequential cells after combinational fault propagation, determined with four fault injection campaigns performing 22.4 million fault experiments each, for which spatial fault configurations are generated at random with a dedicated spatial fault multiplicity m. Numbers are given in millions (10^6).

m	#SEU	#MEU	total	#dSEU	#dMEU
1	8.42 (37.6%)	2.08 (9.3%)	10.50 (46.9%)	3.85 (17.2%)	**0.84 (3.8%)**
2	8.93 (39.9%)	7.08 (31.6%)	16.01 (71.4%)	2.77 (12.4%)	**5.19 (23.2%)**
3	7.07 (31.6%)	11.97 (53.4%)	19.04 (85.0%)	2.49 (11.1%)	**10.19 (45.5%)**
10	0.32 (1.4%)	22.03 (98.3%)	22.36 (99.8%)	0.24 (1.1%)	**21.97 (98.1%)**

Next, I demonstrate the applicability for industrial designs of the proposed method for fault injection in combinational logic, utilizing MiniSAT and FPGA-based fault emulation.

9.3.2 Applicability for Industrial Circuits

In my experiments I wanted to emulate all single faults in combinational and sequential logic for the time frame in which security relevant operations are executed. This way all fault attacks that aim on exploiting single points of attack in circuit *80251* are modeled. Therefore, I used the fault emulator to inject all SEUs in advance, as proposed in Section 8.3.2. Additionally, I injected all distinguishable faults (dMEUs) that were determined after combinational fault propagation of SETs, which were injected into the combinational logic using the software-based pre-processing for enhancing fault injection in combinational logic.

For this purpose, I performed three fault injection campaigns injecting SETs and SEUs in circuit *80251*, as outlined in the columns *#MET*[4] and *#SEU* in Table 9.3. In the first fault injection campaign faults were injected in the CPU, which consists of $N_{\text{FF}} = 5726$ FFs and $N_{\text{C}} = 39,038$ combinational cells, while the circuit was sensitized by test *cpu*. In the second and the third fault injection campaign faults were injected into the crypto-accelerator, which consists of $N_{\text{FF}} = 2775$ FFs and $N_{\text{C}} = 17,603$ combinational cells, while the circuit was sensitized by either test *aes* or test *des*. Circuit *80251-mc* was instrumented for fault injection in sequential logic. The instrumented design including $|L_{\text{FF}}| = N_{\text{FF}} = 8501$ instrumented FFs as well as additional hardware for the emulator's communication interface, fault injection control unit and observation / classification unit fit into 55,526

[4]For simplification, SETs are assumed to be METs with spatial fault multiplicity $m = 1$.

adaptive logic modules (ALMs) on a Stratix II FPGA[5], which corresponds to 77.4% hardware usage in terms of ALMs. For comparison to newer Altera FPGAs, circuit *80251-mc* fits into 138,815 equivalent logic elements (LEs) [Alt06, Alt07]. The design was clocked with $f_{\text{FPGA}} = 26.5\,\text{MHz}$.

For comparison to existing work, I wanted to perform fault injection campaigns using the conventional standalone emulation-based approach for circuit *80251-mc* where fault injection in combinational logic is implemented in hardware on the FPGA using appropriate fault injection cells (refer to Section 6.2.2). This circuit instrumentation technique for combinational logic is similar to the one presented by Kundu et al. [KLPB05], which I expected to generate the least hardware overhead. Still, I could not fit the instrumented circuit *80251-mc* into the FPGA. Therefore, column *speed-up* in Table 9.3 indicates with an ∞ sign that the conventional approach (instrumenting all combinational cells) fails for circuit *80251-mc*.

In contrast, my approach only required to instrument FFs, and therefore, the instrumented circuit fitted into the FPGA. The runtimes of three fault injection campaigns performed for circuit *80251-mc* are listed in column *time* in Table 9.3. Note that the measured runtime includes the time required for the software-based fault injection and propagation of SETs and METs utilizing MiniSAT as well as sequential fault propagation of all SEUs and determined equivalent dMEUs utilizing FPGA-based fault emulation. Columns *#MET* and *#SEU* list the number of injected faults. The three fault injection campaigns for circuit *80251-mc* were executed in about 19 hours, during which in total 60.3 million SETs and 9.1 million SEUs were injected. This demonstrates the applicability of the proposed techniques, whereas conventional stand-alone fault emulation fails. Note that stand-alone software- or simulation-based methods would not be an option either because of the circuit's and tests' complexity.

Fault injection was concentrated to cycles in which cryptographic operations, security operations or the actual operations under test were executed by the CUV, i.e. $N_T < t_{\text{test}}$. For this purpose, the initialization of the circuit, which is steady for all tests and sets the circuit into an initial state, as well as the software-based test result evaluation, which checks the test execution for failures, were excluded from fault injection. This was realized using the feature for observing test-related events, as proposed in Section 6.5.5. This way, unrealistic fault emulation results caused by corrupted initialization or, even worse, corrupted test result evaluation are prevented. Furthermore, this reduces the required time for fault injection campaigns drastically by avoiding

[5]The device EP2S180F1020C3 from the Intel Altera Stratix II family with 71.760 ALMs (179.400 equivalent LEs) was used.

equivalent fault experiments. Potential vulnerabilities during initialization can be analyzed by performing a dedicated fault injection campaign that executes a respective test.

Table 9.3: Total runtime for performing fault injection campaigns, utilizing both FPGA-based fault injection and the software-based pre-processor for faults in combinational logic and covering all possible SEUs and all determined dMEUs caused by SETs ($m = 1$) or METs ($m > 1$). The runtime is given in hour:minute.

circuit	**test**	**#MET**$\cdot 10^6$	**#SEU**$\cdot 10^6$	$\boldsymbol{m}$	**time**	**speed-up**
80251-mc	cpu	35.95	5.27	1	**13:43**	∞
”	aes	15.88	2.50	1	**3:48**	∞
”	des	8.43	1.33	1	**1:45**	∞
8051-mc	isax1	22.40	4.27	1	3:05	**3.64**
”	”	”	”	2	3:36	**3.12**
”	”	”	”	3	4:21	**2.58**
”	”	”	”	10	6:01	**1.87**
8051-mc	isax2	”	”	1	3:51	**5.87**
”	”	”	”	2	5:01	**4.50**
”	”	”	”	3	6:59	**3.23**
”	”	”	”	10	8:00	**2.82**

8051 Microcontroller Since I was not able to fit circuit *80251-mc* into the FPGA when using the conventional emulation-based approach, i.e. instrumenting all combinational cells, I also performed fault injection campaigns for the smaller circuit, circuit *8051-mc*. For these experiments, performance results are discussed and compared to the conventional approach in the next section after comparing the hardware setups of the proposed method and the conventional approach.

Using the proposed method, only FFs need to be instrumented. The instrumented circuit *8051-mc* including $N_{\mathrm{FF}} = 1911$ instrumented FFs and additional hardware for controlling fault injection (performance measures 1 and 2 applied) fit into 16,806 logic elements (LEs) of an Altera Cyclone IV FPGA[6], which corresponds to 11.2% hardware usage in terms of LEs. The FPGA was clocked with $f_{\mathrm{FPGA}} = 20\,\mathrm{MHz}$ in following experiments.

[6]The device EP4CGX150 from the Intel Altera Cyclone IV family with 149,760 LEs was used.

Using the conventional setup, where all $N_\mathrm{C} = 10,029$ combinational cells are instrumented instead, 30,232 LEs are required to fit the instrumented design into the FPGA, i.e. about 79% more LEs compared to the proposed method. Furthermore, because of additional logic in combinational paths, the maximum operating frequency of the conventional setup is limited to $f_\mathrm{FPGA} = 7\,\mathrm{MHz}$, which is a reduction of 69%. The implementation of the conventional approach only allows fault injection in combinational logic, in contrast to my method, which is able to inject faults into FFs as well. Note that instrumenting FFs in addition would considerably increase the hardware overhead and would decrease the operating clock frequency further.

9.3.3 Performance

Circuit *8051-mc* is smaller than *80251-mc*, and therefore, the conventional approach can be applied to it for comparison to existing work. This enabled me to compare the efficiency of the method that I proposed in Chapter 7 for enhancing fault injection in combinational logic to existing work with 1) increasing test durations t_test and 2) increasing fault multiplicities m.

Setup of Fault Injection Campaigns Fault injection campaigns for *8051-mc* were constructed in such a way that these allow to analyze the impact of the test duration and the spatial fault multiplicity on the performance. For this purpose, eight fault injection campaigns were performed, listed in Table 9.3, where four different fault sets were iterated, and either one of the two functional tests *isax1* and *isax2* lasting $t_\mathrm{test} = 11,851$ and $t_\mathrm{test} = 23,811$ clock cycles, respectively, sensitized the circuit.

Although two tests with different durations were used to sensitize the circuit, during each of these fault injection campaigns, 22.40 million SETs or METs as well as 4.27 million SEUs were injected. I chose this setup to keep the number of fault experiments N_e per fault injection campaign constant, allowing to evaluate the impact on the performance of the test duration t_test and the spatial fault multiplicity m. More precisely, for fault injection in combinational logic, I chose the same number of fault injection times $N_T = 2234$ and the same number of spatial fault configurations $|\mathcal{F}_\mathrm{spatial,value}| = N_\mathrm{C} = 10.029$ for all fault injection campaigns. Note that the number of spatial fault configurations matches the number of fault injection locations in combinational logic. For this purpose, I generated sets of spatial-value configurations including $N_\mathrm{C} = 10.029$ spatial configurations in combinational logic each, where I chose a different spatial fault multiplicity $m \in \{ 1, 2, 3, 10 \}$ for each of these sets. Sets of spatial fault configurations with multiplicities $m > 1$ were generated at random, whereas the set for single faults included all possible SETs. Using the proposed software-based preprocessing for fault injection in combinational logic, the sets of spatial fault configurations and all considered fault injection times were iterated. That way, $N_{\mathrm{C}} \cdot N_T = 22.40 \cdot 10^6$ SETs or METs were injected, as listed in column *#MET* of Table 9.3, for which equivalent dMEUs were determined. Then, the determined dMEUs were injected in sequential cells using the FPGA-based fault emulator. Note that METs were generated at random only for benchmark purposes. I suggest to use layout or structural circuit information for multiple fault selection, as presented in [PTH+15, PHB+14, VML+14].

As proposed in Section 8.3.2, I additionally generated all possible single faults in sequential logic (SEUs), so $N_{\mathrm{FF}} = 1911$ spatial-value configurations. During fault generation, I iterated these as well as all considered fault injection times, which resulted in $N_{\mathrm{FF}} \cdot N_T = 4.27 \cdot 10^6$ SEUs, as listed in column *#SEU* of Table 9.3. According to Equation 6.12, $N_e = |\mathcal{F}_{\mathrm{spatial,value}}| \cdot N_T \cdot N_D = (22.40 + 4.27) \cdot 10^6$, where $N_D = 1$ and $N_T < t_{\mathrm{test}}$.

For comparison to existing work, I performed the same fault injection campaigns using the conventional stand-alone emulation-based approach. For fault injection in combinational logic all combinational cells were instrumented, whereas for fault injection in sequential logic all sequential cells were instrumented.

Total Performance The runtime when using the conventional approach was independent of the spatial fault multiplicity and constant with respect to the executed tests. This is due to activating performance optimization measures 1 + 2, as detailed earlier in Section 9.2. Fault injection campaigns for combinational logic using the conventional approach took about 11 minutes when circuit *8051* was sensitized by test *isax1* and 23 minutes when sensitized by test *isax2*.

In contrast, when using the proposed method, the runtime was dependent on both the spatial fault multiplicity and the test duration, as listed in column *time* in Table 9.3. It can be seen, that the performance of the proposed method decreases (increased runtime) for the longer test *isax2* and it also decreases with increasing fault multiplicities. Note that all presented runtimes include the time it takes for combinational fault propagation utilizing MiniSAT as well as sequential fault propagation utilizing FPGA-based fault emulation. Results for SEUs need to be generated only once and could have been reused for following fault injection campaigns, decreasing the runtime of respective fault injection campaigns. Furthermore, sequential fault propagation and combinational fault propagation could have been performed in parallel as soon as the first results are generated by MiniSAT. Note that these optimization possibilities are not considered in Table 9.3 to enable comparison of fault injection campaigns.

As can be seen in Table 9.3, my method performed noticeably better than the conventional approach (*speed-up* 1.9 to 5.9). I noticed that the speed-up of my method is the best for test *isax2*, which lasts about two times longer than test *isax1*, whereas the speed-up was decreased with increasing fault multiplicities. In order to explain the test-dependency and the dependency on the spatial fault multiplicity, I am going to discuss the performance of solely the software-based pre-processing for fault injection in combinational logic, which incorporates MiniSAT.

Performance of MiniSAT Table 9.4 lists the respective times required by MiniSAT for fault injection and performing combinational fault propagation during eight fault injection campaigns for circuit *8051-mc*. Again, each fault injection campaign covered 22.40 million fault injections in combinational logic. Note that MiniSAT was executed in 20 independent instances, which processed equal fractions of to be considered fault sets and were executed by a single Intel(R) Xeon(R) X5660 CPU clocked with 2.80 GHz.

As shown, the time required by MiniSAT was similar in all experiments, although the spatial fault multiplicity was increased. Furthermore, the performance of solely MiniSAT did not depend on the test duration. It was only dependent on the number of iterations required to perform fault injection campaigns. This is reasonable, since fault injection campaigns were constructed such that the same number of $N_\mathrm{C} \cdot N_T = 22.40 \cdot 10^6$ iterations with MiniSAT was required for each fault injection campaign. Although, the effort for injecting faults using MiniSAT's assumption function increases with the spatial fault multiplicity, its contribution to the total runtime is negligible and did not explain the increased runtime for higher fault multiplicities. Remember that more distinguishable faults were generated as the spatial fault multiplicity m was increased, as discussed before with fault propagation results. As a consequence, the efficiency of the proposed performance optimizations decreased since less emulations were skipped, resulting in the performance loss. However, when executing longer tests, longer emulations are skipped, which resulted in a better speed-up compared to the conventional approach. That is, the proposed method performed better for longer tests.

Table 9.4: Time required by MiniSAT for performing fault injection and fault propagation in combinational logic of circuits *80251-mc* and *8051-mc*, which are sensitized by the tests listed in column *test*. The time is given in hour:minute.

circuit	**test**	$m = 1$	$m = 2$	$m = 3$	$m = 10$
8051-mc	isax1	2:22	2:20	2:26	2:23
"	isax2	2:26	2:34	3:13	2:22
80251-mc	cpu	9:08	n.a.	n.a.	n.a.
"	aes	3:33	n.a.	n.a.	n.a.
"	des	1:39	n.a.	n.a.	n.a.

9.4 Summary and Conclusions

I demonstrated the applicability of proposed fault emulation techniques with two different microcontroller designs, varying tests and varying fault sets. Emphasize was put on evaluating proposed performance optimizations. I was able to show that the proposed optimization measures enable fault emulation to provide maximum configurability while providing optimal performance.

Furthermore, I evaluated the method proposed to enhance fault injection in combinational logic, where I was able to show the applicability for an industrial design, for which the conventional stand-alone emulation-based approach failed. In order to compare the performance to existing work, several fault injection campaigns were performed for a smaller circuit, in which a huge number of faults were injected. In total, eight fault injection campaigns for fault injection in combinational logic of circuit *8051-mc* were performed, during which 213 million faults were injected in total. Compared to the conventional approach, the proposed method performed better in all fault injection campaigns with a speed-up of up to factor six.

On top of a good performance, the proposed method enables to model the four fault types SET, MET, SEU and MEU exhaustively, without restricting fault injection to few locations, which makes it suitable for multiple fault selection based on layout or structural information. Moreover, it requires 45% less hardware on FPGAs, increasing applicability for large circuits. Thus, the proposed method for enhancing fault injection in combinational logic constitutes a solution for complex circuits and tests that provides the configurability and the performance required for mimicking fault attacks. Note, the complexity of MiniSAT's propagate function (Boolean constraint propagation without conflicts) is linear with the circuit size and the runtime for sequential fault propagation increases linearly with the test duration. Hence, the proposed fault injection environment is also suitable for even more complex circuits and tests.

Considering the high probability of, e.g., a laser shot to inject multiple faults into a circuit [PTH+15, VML+14], not only security concepts are required to deal with single faults as well as multiple faults with varying fault multiplicities. Also, fault injection tools, such as the presented one, able to handle arbitrary multiple faults efficiently in order to mimic arbitrary fault attacks are required to verify security concepts and their implementations. This allows to maximize the benefit from limited verification times. During my experiments I experienced that single faults injected into combinational logic (SETs) of a circuit that is sensitized by a test may propagate rarely into several hundred sequential cells. That is, during sequential fault propagation utilizing fault emulation equivalent MEUs with fault multiplicities of several hundreds need to be injected. Due to the proposed fault emulation techniques and optimizations, this can be handled with the presented fault emulation without performance loss compared to fault-free test runs.

Chapter 10

Conclusions

In this thesis, I looked into fault injection concepts used to mimic fault attacks in order to verify countermeasures of security circuits. Similar to functional verification it is important to verify the design against security flaws pre-silicon to avoid redesigns, decreasing design costs and time to market. I worked out that for mimicking arbitrary fault attacks it is mandatory to provide the ability to configure arbitrary faults. Unfortunately, the huge variety of combination possibilities for spatial and temporal attack points lead to a fault injection space infeasible to be analyzed completely. Analysis therefore need to focus on subsets of the fault injection space, which have to be selected based on their relevance for the specified security level. Furthermore, respective tools need to provide high performance in order to reduce the required verification time respectively to maximize the security verification coverage within a limited available time for it. Besides configurability and performance, applicability for processor-based security designs is a key requirement. Processor-based designs are usually sensitized with functional tests for security verification, which the corresponding tools need to handle. I propose using FPGA-based fault emulation as pre-silicon fault modeling tool since it fits all these requirements. To remove existing disadvantages and limitations of FPGA-based fault emulation, I proposed new fault emulation techniques in this thesis. My focus was on closing the gap between configurability and performance of FPGA-based fault emulation and to achieve industrial applicability for security related designs.

The proposed fault emulation techniques and implemented fault models are described with a fault configuration model, which I introduced at a meta level. The meta fault configuration model is independent of specific levels of abstraction of a circuit and constitutes a superset of fault models known from literature. It covers all configuration possibilities required to mimic arbitrary fault attacks. For this purpose, a spatial-value configuration space, temporal properties and the respective granularity were defined independently of specific abstraction levels of the circuit under verification. The relation between spatial-value configuration space and temporal properties was described as a fault configuration function, which maps time intervals to arbitrary forcing functions. This way, a meta model was created that has the flexibility to be further specified for specific abstraction levels and fault model types, while it covers any fault configuration relevant to mimic arbitrary physical fault attacks.

Then, I presented an FPGA-based fault emulation environment suitable for modeling multiple faults in sequential cells based on the stuck-at and bit-flip fault models. It supports fault configuration at runtime, which includes configurable fault model types, single and multiple faults and variable fault durations. Fault configurations can be applied at arbitrary discrete times during fault experiments. For fault injection in combinational logic, I presented a new method. It uses a software-based pre-processor, based on Boolean constraint propagation provided by a SAT-solver, to model single and multiple event transients (SETs and METs) and to propagate these to the inputs of sequential cells. Equivalent single and multiple event upsets (SEU and MEU) in sequential cells are then determined and mapped onto the FPGA-based fault emulator for sequential fault propagation. Moreover, equivalence relations between the mapped faults are exploited to drop equivalent faults from further fault emulation. This optimization considerably increased the performance, although an additional pre-processing step is added. In contrast to existing work, the proposed method provides designers a tool to perform extensive fault emulations for SETs, METs, SEUs and MEUs in significantly reduced time. The proposed method for fault injection in combinational logic reduces hardware overhead for circuit instrumentation on FPGAs by 45% and is, hence, still applicable to larger circuits for which conventional approaches require significantly more hardware resources, which may not fit into the available FPGAs. Moreover, the proposed method for fault injection in combinational logic performs up to six times faster for analyzed fault injection campaigns compared to conventional stand-alone FPGA-based fault emulation.

The performance for fault injection in sequential logic was increased with three optimization measures that fight the communication bottleneck of FPGA-based fault emulation. The first two optimization measures do not limit configuration capabilities at all and tend to perform better as the test duration increases. For a wide range of multiple faults the performance is close to the test duration of fault-free fault emulation runs, which is the theoretical optimum. The third measure comes as a trade-off between performance and configurability and provides runtimes close to the theoretical optimum even for very short tests. Additional performance optimization measures for shortening fault experiments and skipping equivalent fault experiments were proposed.

Furthermore, I proposed a feature to support fault configurations with multiple fault injection times, which enables fault emulation to model any fault configuration defined in the total fault injection space in an efficient way without limitations. I presented the first FPGA-based fault emulation environment that provides such powerful configuration possibilities at run time. Thus, I closed the gap between performance and configurability of multiple fault emulation environments. The presented fault emulation environment therefore poses a high performance tool that is suitable for simple as well as sophisticated fault selection strategies and enables to evaluate new ones. It suits fault selection based on layout or structural information and multiple fault injection times can be configured to mimic, e.g., multiple laser fault injection with a time offset.

Further, I proposed a hardware-efficient and performance-efficient concept for evaluating emulation results in hardware. A single fault-free emulation is used to generate a golden reference for following fault emulations. This way, the reference for result comparison has not to be generated by a second instance of the circuit to be verified. Moreover, the fault emulator is sensitive to events on dedicated observation points, which allows to judge the circuit behavior in a more flexible manner and reduces false positives in the emulation results. This method is supported by a self-testing functional test software for processor-based security designs. Based on this feature, temporal fault injection settings can be configured relatively to dedicated events on the defined observation points. For instance, time frames of varying length in which security-relevant operations are executed can be selected for fault injection without manually determining the absolute timing of it.

I demonstrated the applicability of the proposed fault emulation techniques with two different circuits used for security applications, varying tests and varying fault sets, including varying fault multiplicities. Special emphasis was put on evaluating the proposed performance optimizations when increasing the fault multiplicity of the modeled faults. I was able to show that the proposed optimization measures provide fault emulation, on the one hand, a maximum degree of configurability and, on the other hand, feature an optimal performance.

The increased performance, the improved applicability for larger circuits, and the high configurability of FPGA-based fault emulation can be used by the semiconductor industry to analyze more faults in given and typically limited time. This way, the proposed techniques help to find and prevent security flaws in devices that we use in our daily life already during circuit design, i.e. before these are manufactured and shipped to end users respectively fail post silicon certification.

Bibliography

[AAN02] Dan Alexandrescu, Lorena Anghel, and Michael Nicolaidis. New methods for evaluating the impact of single event transients in vdsm ics. In *Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings. 17th IEEE International Symposium on*, pages 99–107. IEEE, 2002.

[ABF94] M. Abramovici, M.A. Breuer, and A.D. Friedman. *Digital Systems Testing and Testable Design.* Electrical engineering, communications and signal processing. IEEE, 1994.

[ABH$^+$02] Christian Aumüller, Peter Bier, Peter Hofreiter, Wieland Fischer, and Jean-Pierre Seifert. Fault attacks on rsa with crt: Concrete results and practical countermeasures. *IACR Cryptology ePrint Archive*, 2002:73, 2002.

[ABTV07] MA Aguirre, V Baena, J Tombs, and Massimo Violante. A new approach to estimate the effect of single event transients in complex circuits. *Nuclear Science, IEEE Transactions on*, 54(4):1018–1024, 2007.

[ADN$^+$10] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: On critical paths and clock faults. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *CARDIS*, volume 6035 of *Lecture Notes in Computer Science*, pages 182–193. Springer, 2010.

[AK96] Ross Anderson and Markus Kuhn. Tamper resistance: A cautionary note. In *Proceedings of the 2Nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[ALF00] Lörinc Antoni, Régis Leveugle, and Béla Fehér. Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes. In *Defect and Fault Tolerance in VLSI Systems*, pages 405–413, 2000.

[ALR01] Algirdas Avizienis, Jean C. Laprie, and Brian Randell. Fundamental Concepts of Dependability. In *Proceedings of the 3rd Information Survivability Workshop*, pages 7–12, Boston, USA, 2001.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.

[Alt06] Altera. FPGA architecture (online). https://www.intel.com/content/dam/us/en/pdfs/literature/wp/wp-01003.pdf, 2006.

[Alt07] Altera. Stratix II device handbook, volume 1 (online). https://www.intel.com/content/dam/us/en/pdfs/literature/hb/stx2/stratix2_handbook.pdf, 2007.

[AZS12] Hamed Abbasitabar, Hamid R. Zarandi, and Ronak Salamat. Susceptibility analysis of LEON3 embedded processor against multiple event transients and upsets. In *CSE, Paphos, Cyprus, December 5-7, 2012*, pages 548–553, 2012.

[Bau05] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, Sept 2005.

[BBC$^+$09] Souheib Baarir, Cécile Braunstein, Renaud Clavel, Emmanuelle Encrenaz, Jean-Michel Ilié, Régis Leveugle, Isabelle Mounier, Laurence Pierre, and Denis Poitrenaud. Complementary formal approaches for dependability analysis. In *24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, DFT 2009, Chicago, Illinois, USA, October 7-9, 2009*, pages 331–339, 2009.

[BCN$^+$04] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *IACR Cryptology ePrint Archive*, 2004:100, 2004.

[BDES14] Bernd Becker, Rolf Drechsler, Stephan Eggersglüß, and Matthias Sauer. Recent advances in sat-based atpg: Non-standard fault models, multi constraints and optimization. In *DTIS*, pages 1–10. IEEE, 2014.

[BDL97] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *Theory and Application of Cryptographic Techniques*, pages 37–51, 1997.

[BDN08] Alberto Bosio and Giorgio Di Natale. LIFTING: A Flexible Open-Source Fault Simulator. In *ATS'08: Asian Test Symposium*, pages 035–040, Saporro, Japan, November 2008.

[BGGG05] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil. Improvement of fault injection techniques based on VHDL code modification. In *IEEE International High-Level Design Validation and Test Workshop*, pages 19–26, 2005.

[BHF97] Vamsi Boppana, Ismed Hartanto, and W. Kent Fuchs. Characterization and implicit identification of sequential indistinguishability. In *VLSI Design*, pages 376–380. IEEE Computer Society, 1997.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential Fault Attacks on Elliptic Curve Cryptosystems. In *International Crytology Conference*, pages 131–146, 2000.

[BS97] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *CRYPTO*, pages 513–525, 1997.

[BS03] Johannes Blömer and Jean-Pierre Seifert. Fault based cryptanalysis of the advanced encryption standard (aes). In Rebecca N. Wright, editor, *Financial Cryptography*, volume 2742 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 2003.

[BSH75] D. Binder, E. C. Smith, and A. B. Holman. Satellite anomalies from galactic cosmic rays. *IEEE Transactions on Nuclear Science*, 22, 1975.

[CDR$^+$16] Stephan De Castro, Jean-Max Dutertre, Bruno Rouzeyre, Giorgio Di Natale, and Marie-Lise Flottes. Frontside versus backside laser injection: A comparative study. *J. Emerg. Technol. Comput. Syst.*, 13(1):7:1–7:15, November 2016.

[CET13] Liang Chen, Mojtaba Ebrahimi, and Mehdi Baradaran Tahoori. CEP: correlated error propagation for hierarchical soft error analysis. *J. Electronic Testing*, 29(2):143–158, 2013.

[CHD95] Kwang-Ting Cheng, Shi-Yu Huang, and Wei-Jin Dai. Fault emulation: a new approach to fault grading. In *International Conference on Computer Aided Design*, pages 681–686, 1995.

[CHD99] Kwang-Ting Cheng, Shi-Yu Huang, and Wei-Jin Dai. Fault emulation: A new methodology for fault grading. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 18:1487–1495, 1999.

[CMD01] Li Chen, Student Member, and Sujit Dey. Software-based self-testing methodology for processor cores. *IEEE Trans. Computer-Aided Design*, 20:369–380, 2001.

[CMR$^+$01] Pierluigi Civera, Luca Macchiarulo, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Exploiting FPGA for Accelerating Fault Injection Experiments. In *International On-Line Testing Symposium*, pages 9–13, 2001.

[CMR$^+$02] Pierluigi Civera, Luca Macchiarulo, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. An FPGA-Based Approach for Speeding-Up Fault Injection Campaigns on Safety-Critical Circuits. *Journal of Electronic Testing*, 18:261–271, 2002.

[Com12] Common Criteria. Common criteria for information technology security evaluation. Technical report, Common Criteria, September 2012.

[Con03] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *Micro, IEEE*, 23(4):14–19, July 2003.

[CP95] Jeffrey A. Clark and Dhiraj K. Pradhan. Fault Injection: A Method for Validating Computer-System Dependability. *IEEE Computer*, 28:47–56, 1995.

[DBG+09] J. M. Daveau, A. Blampey, G. Gasiot, J. Bulone, and P. Roche. An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip. In *2009 IEEE International Reliability Physics Symposium*, pages 212–220, April 2009.

[DDRT12] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of AES. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*, pages 7–15, 2012.

[Dia75] Francisco J. O. Dias. Fault masking in combinational logic circuits. *IEEE Trans. Computers*, 24(5):476–482, 1975.

[DLV03] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on a.e.s. *IACR Cryptology ePrint Archive*, 2003:10, 2003.

[DNFLR12] Giorgio Di Natale, Marie-Lise Flottes, Feng Lu, and Bruno Rouzeyre. tLIFTING : A Multi-level Delay-annotated Fault Simulator for Digital Circuits. In *DCIS'2012: XVII Conference on Design of Circuits and Integrated Systems*, page 1, November 2012. Poster.

[EAT13] Mojtaba Ebrahimi, Hossein Asadi, and Mehdi Baradaran Tahoori. A layout-based approach for multiple event transient analysis. In *DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 100:1–100:6, 2013.

[EMEM14] Mojtaba Ebrahimi, Abbas Mohammadi, Alireza Ejlali, and Seyed Ghassem Miremadi. A fast, flexible, and easy-to-develop fpga-based fault injection technique. *Microelectronics Reliability*, 54(5):1000–1008, 2014.

[ERTU07] Peeter Ellervee, Jam Raik, K. Tammemäe, and Raimund Ubar. FPGA-based fault emulation of synchronous sequential circuits. *Iet Computers and Digital Techniques*, 1, 2007.

[ES] Niklas Eén and Niklas Sörensson. The MiniSAT Page (online). http://www.minisat.se/.

[ESRT15] Mojtaba Ebrahimi, Nour Sayed, Maryam Rashvand, and Mehdi Baradaran Tahoori. Fault injection acceleration by architectural importance sampling. In *2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, Netherlands, October 4-9, 2015*, pages 212–219, 2015.

[EVC$^+$09] Luis Entrena, Mario García Valderas, Raúl Fernández Cardenal, Marta Portela Garcia, and Celia López Ongil. Set emulation considering electrical masking effects. *IEEE Transactions on Nuclear Science*, 56(4):2021–2025, 2009.

[FT09] Toshinori Fukunaga and Junko Takahashi. Practical fault attack on a cryptographic lsi with iso/iec 18033-3 block ciphers. In Luca Breveglieri, Israel Koren, David Naccache, Elisabeth Oswald, and Jean-Pierre Seifert, editors, *FDTC*, pages 84–92. IEEE Computer Society, 2009.

[Gai97] Jiri Gaisler. Evaluation of a 32-bit microprocessor with built-in concurrent error-detection. In *FTCS*, pages 42–46. IEEE Computer Society, 1997.

[Gar15] Gartner, Inc. Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015 (online). http://www.gartner.com/newsroom/id/3165317, 2015.

[Gar17a] Gartner, Inc. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016 (online). http://www.gartner.com/newsroom/id/3598917, 2017.

[Gar17b] Gartner, Inc. Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016 (online). http://www.gartner.com/newsroom/id/3609817, 2017.

[GC91] Sumit Ghosh and Tapan J. Chakraborty. On behavior fault modeling for digital designs. *Journal of Electronic Testing*, 2:135–151, 1991.

[Gir03] Christophe Giraud. Dfa on aes. *IACR Cryptology ePrint Archive*, 2003:8, 2003.

[GJKC06] Rajesh Garg, Nikhil Jayakumar, Sunil P. Khatri, and Gwan Choi. A design approach for radiation-hard digital electronics. In Ellen Sentovich, editor, *DAC*, pages 773–778. ACM, 2006.

[GKS+11a] Johannes Grinschgl, Armin Krieg, Christian Steger, Reinhold Weiss, Holger Bock, and Josef Haid. Automatic saboteur placement for emulation-based multi-bit fault injection. In *International Workshop on Reconfigurable Communication-Centric Systems-on-Chip*, 2011.

[GKS+11b] Johannes Grinschgl, Armin Krieg, Christian Steger, Reinhold Weiss, Holger Bock, and Josef Haid. Modular Fault Injector for Multiple Fault Dependability and Security Evaluations. In *Euromicro Symposium on Digital Systems Design*, 2011.

[GT04] Christophe Giraud and Hugues Thiebeauld. A survey on fault attacks. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kalam, editors, *CARDIS*, volume 153 of *IFIP*, pages 159–176. Kluwer/Springer, 2004.

[Hal14] Neelesh Halinge. Performance optimization of a fault emulation environment, 2014.

[Hen61] F. C. Hennie. *Iterative Arrays of Logical Circuits.* MIT Press, Cambridge Mass., 1961.

[Jan09] Stephan Janssen. Erweiterung einer 8051 CPU um digitale Fehlererkennungsmaßnahmen, 2009.

[JAR+94] Eric Jenn, Jean Arlat, Marcus Rimén, Joakim Ohlsson, and Johan Karlsson. Fault Injection into VHDL Models: The MEFISTO Tool. In *Symposium on Fault-Tolerant Computing*, pages 66–75, 1994.

[JHSS11] Angelika Janning, Johann Heyszl, Frederic Stumpf, and Georg Sigl. A cost-effective fpga-based fault simulation environment. *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 21–31, 2011.

[KE15] Hans G. Kerkhoff and Hassan Ebrahimi. Intermittent resistive faults in digital cmos circuits. In Zoran Stamenkovic, Witold A. Pleskacz, Jaan Raik, and Heinrich Theodor Vierhaus, editors, *DDECS*, pages 211–216. IEEE, 2015.

[KHP04] Tanay Karnik, Peter Hazucha, and Jagdish Patel. Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. *IEEE Transactions on Dependable and Secure Computing*, 1:128–143, 2004.

[KJP14] Raghavan Kumar, Philipp Jovanovic, and Ilia Polian. Precise fault-injections using voltage and temperature manipulation for differential cryptanalysis. In *IOLTS*, pages 43–48. IEEE, 2014.

[KKJ10] Zvi Kohavi and Niraj K. Jha. *Switching and Finite Automata Theory.* Cambridge University Press, 3rd edition, 2010.

[KLPB05] Sandip Kundu, Matthew DT Lewis, Ilia Polian, and Bernd Becker. A soft error emulation system for logic circuits. In *DCIS*, 2005.

[KP74] S. Kamal and C. V. Page. Intermittent Faults: A Model and a Detection Procedure. *IEEE Transactions on Computers*, C-23:713–719, 1974.

[KRL15] Bradley T. Kiddie, William H. Robinson, and Daniel B. Limbrick. Single-event multiple-transient characterization and mitigation via alternative standard cell placement methods. *ACM Trans. Des. Autom. Electron. Syst.*, 20(4):60:1–60:22, September 2015.

[LCMV09] Régis Leveugle, A. Calvez, Paolo Maistri, and Pierre Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *DATE, Nice, France, April 20-24, 2009*, pages 502–506, 2009.

[LDCDN$^+$15] Feng Lu, Stephan De Castro, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. Dynamic fault model for long duration laser-induced fault simulation. In *TRUDEVICE Workshop, Grenoble, France, March*, 2015.

[LDJK94] Peter Lidén, Peter Dahlgren, Rolf Johansson, and Johan Karlsson. On latching probability of particle induced transients in combinational networks. In *FTCS*, pages 340–349. IEEE Computer Society, 1994.

[Lev00] Régis Leveugle. Fault Injection in VHDL Descriptions and Emulation. In *Defect and Fault Tolerance in VLSI Systems*, pages 414–419, 2000.

[Lev05] Régis Leveugle. A new approach for early dependability evaluation based on formal property checking and controlled mutations. In *11th IEEE International On-Line Testing Sym-*

posium (IOLTS 2005), 6-8 July 2005, Saint Raphael, France, pages 260–265, 2005.

[Lev07] Régis Leveugle. Early analysis of fault-based attack effects in secure circuits. *IEEE Trans. Computers*, 56(10):1431–1434, 2007.

[LGPE05] Celia López-Ongil, Mario García-Valderas, Marta Portela-García, and Luis Entrena-Arrontes. Autonomous Transient Fault Emulation on FPGAs for Accelerating Fault Grading. In *International On-Line Testing Symposium*, pages 43–48, 2005.

[LGPE07] Celia López-Ongil, Mario García-Valderas, Marta Portela-García, and Luis Entrena-Arrontes. Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation. *IEEE Transactions on Nuclear Science*, 54:252–261, 2007.

[LH92] H. K. Lee and D. S. Ha. Hope: An efficient parallel fault simulator for synchronous sequential circuits. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC '92, pages 336–340, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[LH07] Daniel Larsson and Reiner Hähnle. Symbolic fault injection. In Bernhard Beckert, editor, *Proc. 4th International Verification Workshop (Verify) in connection with CADE-21 Bremen, Germany*, volume 259, pages 85–103. CEUR Workshop Proceedings, 2007.

[LNF$^+$14] Feng Lu, Giorgio Di Natale, Marie-Lise Flottes, Bruno Rouzeyre, and Guillaume Hubert. Layout-aware laser fault injection simulation and modeling: From physical level to gate level. In *DTIS*, pages 1–6. IEEE, 2014.

[May78] *A New Physical Mechanism for Soft Errors in Dynamic Memories*, May 1978.

[MDH$^+$13] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, pages 77–88, 2013.

[MDH$^+$14] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. *CoRR*, abs/1402.6421, 2014.

[MEEM12] Abbas Mohammadi, Mojtaba Ebrahimi, Alireza Ejlali, and Seyed Ghassem Miremadi. SCFIT: A fpga-based fault injection technique for seu fault model. In Wolfgang Rosenstiel and Lothar Thiele, editors, *DATE*, pages 586–589. IEEE, 2012.

[MLB$^+$14] Paolo Maistri, Régis Leveugle, Lilian Bossuet, A. Aubert, Viktor Fischer, Bruno Robisson, Nicolas Moro, Philippe Maurine, Jean-Max Dutertre, and Mathieu Lisart. Electromagnetic analysis and fault injection onto secure circuits. In *22nd International Conference on Very Large Scale Integration, VLSI-SoC, Playa del Carmen, Mexico, October 6-8, 2014*, pages 1–6, 2014.

[MM07] Natasa Miskov-Zivanov and Diana Marculescu. Soft error rate analysis for sequential circuits. In *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*, pages 1436–1441, 2007.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards.* Springer, 2007.

[MVS08] Silvio Misera, Heinrich Theodor Vierhaus, and André Sieber. Simulated fault injections and their acceleration in systemc. *Microprocessors and Microsystems - Embedded Hardware Design*, 32:270–278, 2008.

[MZM10] Natasa Miskov-Zivanov and Diana Marculescu. Multiple transient faults in combinational and sequential circuits: A systematic approach. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 29(10):1614–1627, 2010.

[NHHS16] Ralph Nyberg, Johann Heyszl, Dietmar Heinz, and Georg Sigl. Enhancing fault emulation of transient faults by separating combinational and sequential fault propagation. In Ayse Kivilcim Coskun, Martin Margala, Laleh Behjat, and Jie Han, editors, *ACM Great Lakes Symposium on VLSI*, pages 209–214. ACM, 2016.

[NHN$^+$14] Ralph Nyberg, Johann Heyszl, Jürgen Nolles, Dirk Rabe, and Georg Sigl. Closing the Gap Between Speed and Configurability of Multi-Bit Fault Emulation Environments for Security and Safety-Critical Designs. In *Euromicro Symposium on Digital Systems Design*, pages 114–121, 2014.

[NHRS15] Ralph Nyberg, Johann Heyszl, Dirk Rabe, and Georg Sigl. Closing the gap between speed and configurability of multi-bit fault emulation environments for security and safety–critical designs. *Microprocessors and Microsystems*, May 2015.

[NHS15] Ralph Nyberg, Johann Heyszl, and Georg Sigl. Efficient fault emulation through splitting combinational and sequential fault propagation. In *1st International Workshop on Resiliency in Embedded Electronic*, 2015.

[Nic10] Michael Nicolaidis. *Soft Errors in Modern Electronic Systems.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[NJJ06] André K. Nieuwland, Samir Jasarevic, and Goran Jerin. Combinational logic soft error analysis and protection. In *IOLTS*, pages 99–104. IEEE Computer Society, 2006.

[NR11] Ralph Nyberg and Dirk Rabe. Verifikation von Fehlererkennungsmaßnahmen in Security-Controllern per Emulation. In *DACH Security 2011.* Peter Schartner and Jürgen Taeger, 2011.

[NtCP92] Thomas M. Niermann, Wu tung Cheng, and Janak H. Patel. PROOFS: a fast, memory-efficient sequential circuit fault simulator. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 11:198–207, 1992.

[OGST$^+$14] Sébastien Ordas, Ludovic Guillaume-Sage, Karim Tobich, Jean-Max Dutertre, and Philippe Maurine. Evidence of a larger em-induced fault model. In Marc Joye and Amir Moradi, editors, *CARDIS*, volume 8968 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2014.

[Ore02] Oregano Systems Design & Consulting GesmbH. MC8051 IP Core Synthesizeable VHDL Microcontroller IP-Core User Guide, 2002.

[PGSR10] Mihalis Psarakis, Dimitris Gizopoulos, Edgar E. Sánchez, and Matteo Sonza Reorda. Microprocessor software-based self-testing. *IEEE Design & Test of Computers*, 27(3):4–19, 2010.

[PHB$^+$14] Athanasios Papadimitriou, David Hély, Vincent Beroulle, Paolo Maistri, and Régis Leveugle. A multiple fault injection methodology based on cone partitioning towards RTL modeling of laser attacks. In *DATE, Dresden, Germany, March 24-28*, pages 1–4, 2014.

[PHRB11] Ilia Polian, John P. Hayes, Sudhakar M. Reddy, and Bernd Becker. Modeling and Mitigating Transient Errors in Logic Circuits. *IEEE Transactions on Dependable and Secure Computing*, 8:537–547, 2011.

[PKE$^+$11] Samuel Pagliarini, Fernanda Kastensmidt, Luis Entrena, Almudena Lindoso, and Enrique San Millan. Analyzing the Impact of Single-Event-Induced Charge Sharing in Complex Circuits. *IEEE Transactions on Nuclear Science*, 58:2768–2775, 2011.

[PNKI13] Karthik Pattabiraman, Nithin Nakka, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Symplfied: Symbolic program-level fault injection and error detection framework. *IEEE Trans. Computers*, 62(11):2292–2307, 2013.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.

[PSC$^+$12] Andrea Pellegrini, Robert Smolinski, Lei Chen, Xin Fu, Siva Kumar Sastry Hari, Junhao Jiang, Sarita V. Adve, Todd M. Austin, and Valeria Bertacco. Crashtest'ing swat: Accurate, gate-level evaluation of symptom-based resiliency solutions. In Wolfgang Rosenstiel and Lothar Thiele, editors, *DATE*, pages 1106–1109. IEEE, 2012.

[PTH$^+$15] Athanasios Papadimitriou, Marios Tampas, David Hély, Vincent Beroulle, Paolo Maistri, and Régis Leveugle. Validation of RTL laser fault injection model with respect to layout information. In *HOST, Washington, DC, USA, 5-7 May*, pages 78–81, 2015.

[QS02] Jean-Jacques Quisquater and David Samyde. Eddy current for Magnetic Analysis with Active Sensor. In *Esmart 2002, Nice, France*, 9 2002.

[RCS$^+$15] Andreas Riefert, Riccardo Cantoro, Matthias Sauer, Matteo Sonza Reorda, and Bernd Becker. On the automatic generation of sbst test programs for in-field test. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 1186–1191, 2015.

[Reo15] Matteo Sonza Reorda. In-field test of safety-critical systems: is functional test a feasible solution? In *16th Latin-American Test Symposium, LATS 2015, Puerto Vallarta, Mexico, March 25-27, 2015*, pages 1–2, 2015.

[RETU05] Jaan Raik, Peeter Ellervee, Valentin Tihhomirov, and Raimund Ubar. Improved Fault Emulation for Synchronous Sequential Circuits. In *Euromicro Symposium on Digital Systems Design*, pages 72–78, 2005.

[RSDT13] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault model analysis of laser-induced faults in sram memory cells. In Wieland Fischer and Jörn-Marc Schmidt, editors, *FDTC*, pages 89–98. IEEE Computer Society, 2013.

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In *CHES*, pages 2–12, 2002.

[SBHS15] Bodo Selmke, Steffan Brummer, Johann Heyszl, and Georg Sigl. Precise laser fault injections into 90 nm and 45 nm sram-cells. In *CARDIS, Bochum, Germany, November 4-6, 2015*, 2015.

[SBP15] Matthias Sauer, Bernd Becker, and Ilia Polian. Phaeton: A SAT-based framework for timing-aware path sensitization (pre-print). In *IEEE Transactions on Computers*. IEEE, (accepted) 2015.

[SGD08] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. Practical setup time violation attacks on AES. In *Seventh European Dependable Computing Conference, EDCC-7 2008, Kaunas, Lithuania, 7-9 May 2008*, pages 91–96, 2008.

[SH07] Jörn-Marc Schmidt and Michael Hutter. Optical and em fault-attacks on crt-based rsa: Concrete results. In Johannes Wolkerstorfer Karl C. Posch, editor, *Austrochip 2007, 15th Austrian Workhop on Microelectronics, 11 October 2007, Graz, Austria, Proceedings*, pages 61 – 67. Verlag der Technischen Universität Graz, 2007.

[Shi00] R. Shirey. Internet security glossary. RFC 2828, The Internet Society, May 2000.

[Sko05] Sergei P. Skorobogatov. Semi-invasive attacks – a new approach to hardware security analysis, 2005.

[Sko09] Sergei P. Skorobogatov. Local heating attacks on flash memory devices. In Mohammad Tehranipoor and Jim Plusquellic, editors, *HOST*, pages 1–6. IEEE Computer Society, 2009.

[SMS13] Mohammad Shokrolah Shirazi, Brendan Morris, and Henry Selvaraj. Fast fpga-based fault injection tool for embedded processors. In *International Symposium on Quality Electronic Design, ISQED 2013, Santa Clara, CA, USA, March 4-6, 2013*, pages 476–480, 2013.

[ST$^+$11] Matthias Sauer, Victor Tomashevich, Jörg Müller 0004, Matthew D. T. Lewis, A. Spilla, Ilia Polian, Bernd Becker, and Wolfram Burgard. An fpga-based framework for run-time injection and analysis of soft errors in microprocessors. In *IOLTS*, pages 182–185. IEEE, 2011.

[TK10] Elena Trichina and Roman Korkikyan. Multi fault laser attacks on protected crt-rsa. In Luca Breveglieri, Marc Joye, Israel Koren, David Naccache, and Ingrid Verbauwhede, editors, *FDTC*, pages 75–86. IEEE Computer Society, 2010.

[TMS$^+$13] Nikolaus Theißing, Dominik Merli, Michael Smola, Frederic Stumpf, and Georg Sigl. Comprehensive analysis of software countermeasures against fault attacks. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 404–409, 2013.

[VLR06] Pierre Vanhauwaert, Régis Leveugle, and Philippe Roche. Reduced Instrumentation and Optimized Fault Injection Control for Dependability Analysis. In *Very Large Scale Integration*, pages 391–396, 2006.

[VML$^+$14] Pierre Vanhauwaert, Paolo Maistri, Régis Leveugle, Athanasios Papadimitriou, David Hély, and Vincent Beroulle. On error models for RTL security evaluations. In *DTIS 2014, Santorini, Greece, May 6-8*, pages 1–6, 2014.

[WX11] Feng Wang and Yuan Xie. Soft error rate analysis for combinational logic using an accurate electrical masking model. *IEEE Trans. Dependable Sec. Comput.*, 8(1):137–146, 2011.

[Zie96] James F. Ziegler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1):19–40, 1996.

[ZMM$^+$06] Ming Zhang, Subhasish Mitra, T. M. Mak, Norbert Seifert, Nicholas J. Wang, Quan Shi, Kee Sup Kim, Naresh R. Shanbhag, and Sanjay J. Patel. Sequential element design with built-in soft error resilience. *IEEE Trans. VLSI Syst.*, 14(12):1368–1378, 2006.