Hardware Simulation of Embedded Software Fault Attacks: How to SimpliFI Processor Fault Vulnerability Evaluation

by

Jacob T. Grycel

A Thesis Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical and Computer Engineering

May 2021

APPROVED:

_____

Dr. Patrick Schaumont, Major Thesis Advisor

_____

Dr. Berk Sunar, Committee Member

_____

Dr. Shahin Tajik, Committee Member

Abstract

Physical attacks on hardware are increasingly becoming a major consideration in the design of embedded and digital systems. Although side-channel analysis attacks are generally understood, even in complex embedded processors, embedded software vulnerabilities to fault attacks are hard to predict. Despite knowing general behaviors that may be caused by fault attacks, the multiple levels of abstraction between software and the physical hardware make it challenging to predict precisely how a piece of software will respond to fault injection attacks. Current fault evaluation methodologies are split along the hardware/software divide, with hardware fault analysis techniques focusing on simulating all possible faults with no ties to the software, and software simulation methods trading physical accuracy for software state tracking. As a result, the burden of collecting realistic data on software fault vulnerabilities falls on physical device tests, missing the goal of performing fault evaluation during the hardware design cycle. This thesis presents SimpliFI, a methodology for evaluating software fault vulnerabilities and exploring their root causes with realistic fault behavior captured at the hardware level. SimpliFI captures software-level fault effects by simulating instructions being executed by the processor, and hardware-level fault propagation by observing gate-level hardware simulation data. This simulation framework is first defined using broad requirements in order to be applicable to a wide range of devices, and then implemented for a RISC-V embedded processor. The results collected using SimpliFI provide insight into the root cause of software instruction fault responses, and determine realistic faulty outputs of attacks on larger applications.

Acknowledgments

I would like to express my deepest gratitude to my advisor Professor Patrick Schaumont for his guidance, insight, and support throughout my research and additional projects this year. Beyond his role as my research advisor, he was extremely supportive as a person and made a challenging year more manageable; I am truly grateful for this.

I would also like to thank my committee members for their time and support reviewing my thesis. I am grateful for Professor Berk Sunar's support as my undergraduate Major Qualifying Project advisor, and for the enlightening discussions with Professor Shahin Tajik during group meetings this year.

My sincerest thanks go to Dan Walters, Joe Chapman, and Rachel Bainbridge, my mentors at The MITRE Corporation, who have challenged me to develop new skills and ways of thinking about embedded security and engineering over the last four years. My project experiences with them added an entire second dimension to my academic experience I would have not found elsewhere.

I thank Professor Robert Walls for his support and generosity as my research advisor throughout my undergraduate studies. I would not be where I am today without the opportunities and encouragement he graciously offered me.

Finally, I would like to thank my family and friends for their unending love, support, and compassion throughout my entire life. I have been lucky to have such wonderful people around me, and I make sure to never lose sight of that.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cybersecurity has risen as a major focus area in computing since "The Creeper", the first known computer virus, was released on the ARPANET in 1971 [2]. Though major technological advances in computer technology are driven by government and consumer product development, society's widespread adoption of computers has pushed cybersecurity into the spotlight as a critical area of concern. With nearly every aspect of modern life tied to some sort of computer system, whether it be a device or server, the protection of peoples' information and privacy is continually being seen as a larger priority.

The sheer volume of network-enabled, daily-use software pushes focus onto software and network security in order to prevent remote attackers from gaining access to a computing system. The resulting countermeasures and attack knowledge has led to numerous, widely-adopted security standards for applications and network protocols. However, these are not the only fronts for malicious cyber activity.

Cybersecurity at the software and network level is extremely visible because we interact with it every day. Yet, none of our modern software and connectivity would be possible without sophisticated hardware. The underlying hardware is responsible for performing real computations, rendering graphics, and managing network connections. All of these critical operations make hardware an appealing target for knowledgeable adversaries that want to strike at the heart of a system. Obtaining physical devices to perform hardware attacks was traditionally more difficult due to the devices of interest residing in facilities with limited access. But now, many small devices out in the world are connected to larger information systems, making hardware attacks more feasible than ever.

Two well-studied forms of physical attacks on hardware are side-channel analysis (SCA) and fault injection attacks. Side-channel attacks leverage information leakages from opera-

tions on sensitive data, often in the form of power consumption or electromagnetic emanation analysis. Fault attacks instead involve physical interference with the device to disrupt critical execution steps and induce unintended behavior. In 1996, the Bellcore attack, the first published form of cryptographic fault attack, demonstrated that an RSA encryption key could be recovered with differential analysis (DFA) applied to corrupted outputs caused by naturally-occurring hardware faults [3]. A year later, Biham and Shamir expanded the capabilities of DFA to apply to secret-key algorithms [4]. Since then, there has been extensive research conducted to develop our understanding of fault attacks and develop countermeasures to prevent or mitigate them. With fault attacks having been demonstrated on both embedded software and pure hardware designs, tools for evaluating fault vulnerabilities before production could be an integral part of the embedded systems development cycle.

Evaluating a system of hardware and software components for fault vulnerabilities requires analysis at many levels ranging from device-level physics to software-level error propagation. At the hardware level, faults can be injected into a system by violating assumed operating parameters, such as electrical signal timing. Evaluating the effects of fault injection techniques on hardware requires a precise understanding of how clock glitches, voltage manipulations, electromagnetic field pulses, and other injection mechanisms induce faulty bits in the digital circuitry. Meanwhile, software-level evaluation is concerned with analyzing how faults caused by hardware level injection techniques impact the program as a whole. The effects on software can be anything from instruction skips to corrupted program outputs.

However, these distinct goals of evaluating hardware vs software have led to disjointed evaluation methodologies. Evaluation methodologies for hardware-level fault effects focus on determining the precise behavior of a fault propagating through the digital circuitry [5]. The difficulty of accurately modeling how a fault affects device physics has led to few methodologies actually considering realistic fault manifestation [6]. Meanwhile, evaluation methodologies for software-level fault effects forfeit physical accuracy in return for more efficient software data and control flow analysis [7]. The only current methodology for evaluating physically accurate fault attacks at the software level involves performing attacks on real devices. Since fault responses are highly device-dependent, the results from physical testing on one device will not hold entirely true for engineers developing new platforms and software.

To split the gap between hardware and software fault evaluation, this thesis introduces the SImulation Methodology for embedded Processors to Learn the Impacts of Fault Injection (SimpliFI) and demonstrates how hardware simulation can be used to obtain physically-accurate software fault evaluation results. This approach uses physical attributes of an actual device, such as gate layout and signal propagation, to predict realistic responses to fault injection attacks. By encapsulating hardware fault propagation and software execution in one simulation, SimpliFI supports root cause analysis of software fault impacts for both short instruction sequences and full applications. To the best of our knowledge, SimpliFI is the first design-time methodology in the public domain that enables automated fault

evaluation and explains software-level fault effects through their manifestation in the device microarchitecture.

This thesis presents SimpliFI as a high-level generic framework for automatic evaluation of embedded software fault vulnerabilities, and presents an implementation of the framework for the open-source BRISC-V embedded processor [8, 9, 10]. The results collected by the framework are able to show how different fault injection parameters affect the processor state, which faults propagate to program outputs, and which subsets of the processor microarchitecture are more susceptible to fault injection. Through developing the framework, building its implementation, and analyzing its results, this thesis presents the additional following contributions:

- Scripting methods of duplicating a post-layout netlist to obtain identical device images with different programs loaded.
- Techniques for instrumenting C code with unique identifiers for targeting simulated fault injection points.
- Programmatic methods for automating multiple hardware simulations with different runtime parameters within one invocation of the simulator.
- A scripting method for recompiling standard Xilinx simulation libraries to simulate a simple metastability model.
- A custom file format that allows a user to plan multiple fault injection tests automated in the SimpliFI simulation framework.
- Expanded categories for organizing embedded software fault attack outcomes.
- Visualizations of hardware fault propagation for multiple fault injections and target points.

The remainder of this thesis is structured as follows. Chapter 2 discusses background topics in fault injection techniques, digital circuit timing and signal propagation, and current fault injection analysis methods. Chapter 3 presents SimpliFI as a generic framework that supports specific functional requirements. Chapter 4 discusses a practical implementation of the framework using the Xilinx Vivado Design Suite to analyze a BRISC-V FPGA core. Chapter 5 discusses how results from SimpliFI provide insight into the fault vulnerabilities of embedded software running on BRISC-V. Finally, Chapter 6 concludes the thesis by discussing SimpliFI's place in the security electronic design automation tool landscape and proposing useful extensions for future work.

Chapter 2

Background and Current Techniques

A fault vulnerability framework, such as the one presented in this thesis, is only now possible after years of industry and academic research in fault injection attacks. In contrast to their sibling subject area, side-channel analysis, practical fault injection attacks are highly dependent on physical properties of a target device. The success of embedded software fault injection attacks is impacted by the following details: type of fault injection; fault injection parameters; device microarchitecture; target application; physical circuit characteristics; operating conditions.

This chapter builds an understanding of how these different attack aspects are relevant for designing an accurate fault vulnerability evaluation tool. The first part discusses pertinent information about digital circuit design necessary for analyzing fault behavior. The second part provides both an overview of embedded processors and instruction set architectures. The third part provides an introduction to fault injection, an overview of different fault injection techniques, and a discussion of challenges in fault vulnerability evaluation. The fourth part discusses current knowledge and methods for understanding how fault attacks affect embedded software and hardware.

2.1 Digital Circuits

2.1.1 Circuit Timing

Digital circuits, whether fabricated as an Application Specific Integrated Circuit (ASIC), or programmed in a Field-Programmable Gate Array (FPGA), comprise a network of combina-

torial gates and sequential elements. Combinatorial gates implement a logic function, such as AND, OR, or XOR. These types of gates do not store any value, and instead constantly update their output to represent the correct function of the input signals. On the other hand, sequential elements store and transfer binary values, but do not perform any logic functions on the data. The transfer of data from one sequential element (*register*) to another is synchronized to a time-keeping signal called the *clock*. The clock is typically a 50% duty-cycle square wave oscillating between high and low logic levels. Registers are designed so that when the clock has a transition from logic 0 to logic 1 (*rising edge*), the data on the input is stored and transmitted on the output. This data transfer happens on every rising edge of the clock to perform sequential computing.

While it is easy to think of combinatorial gates as immediately computing the result of the inputs, the reality is that all data transfers and updates take a non-zero amount of time. All digital circuit elements are built from some type of transistor circuit, usually complementary metal-oxide-semiconductor (CMOS) devices, which themselves have signal propagation delays from when an input changes to when the output updates. Because of these physical delays, signal timing is a critical aspect of correct digital circuit operation.

When a large network of combinatorial gates are connected together, the amount of time it takes for first-level input to propagate to the final output increases. Furthermore, this propagation delay may vary in time depending on the input combination and which input values have changed. Much like the propagation delay through a single gate, electrical signals in the circuit also take time to travel along the connecting circuit wires. As the circuit is being designed and physically laid out, any changes made to the connecting wire lengths will result in timing changes. In a combinatorial network in between a set of destination and source registers, the time in between two clock edges must be long enough for the longest possible propagation delay to occur. Therefore, the period of the clock signal is restricted by longest combinatorial path in the entire circuit. The longest delay path between two registers is called the *critical path*.

In addition to the gate and routing delay, registers, such as flip-flops, have further timing requirements. Registers have "clock to Q" delays due to the time it takes for the sampled input value to propagate to the register output. Registers also have timing parameters called the *setup* and *hold* times. The setup time defines the amount of time before the incoming clock edge where the input data must be stable and not change. The hold time defines the amount of time after the clock edge for which the input data must still remain stable. Hold violations usually appear in the form of paths that have such short delay that a new value propagates to the register before the hold time passes. Finally, since the clock signal itself has transmission delays across a circuit, the clock edge arrives with a small but non-zero time offset from the intended transition point, called the *clock skew*. Equation 2.1 incorporates all of the circuit timing components into two inequalities that must hold to ensure correct circuit operation.

$$\tau > t_{CQ} + t_{logic} + t_{setup} + t_{skew} \tag{2.1a}$$
$$t_{hold} < t_{CQ} + t_{tlogic} \tag{2.1b}$$

where

τ is the clock period
t_{CQ} is the register clock-to-Q delay
t_{logic} is the critical path of the circuit
t_{setup} is the register setup time
t_{skew} is the clock skew
t_{hold} is the register hold time

2.1.2 Metastability

When the data input to a register violates the setup time, the register can be forced into a metastable state where the output does not settle to a 1 or 0 within the normal propagation time. The resolution of a setup time violation is a function of the initial state of the data input, the transition time of the data input, and the exact placement of the sampling point within the register sample window [11, 12]. According to the models and experiments by Horstmann et al., the final register value is high-dependent on the amplitude of the data signal and the switching time of the flip-flop bistability circuit [13].

If the data voltage is close to its initial or final state when the register input switches off, the output value will resolve to a logic 0 or 1. However, if the voltage is close to the middle of the positive and negative supplies, the register is likely to end up in an invalid logic state in between 0 and 1. When this happens, signal noise in low-level components can either push the register to a valid state, or keep it in the metastable state. If the register successfully resolves during a setup time violation the output propagation time of the register can take significantly longer than during normal operation [13]. If the register does not resolve due to environmental noise, the metastable state behavior is dependent on the underlying technology. Flip-flops built from CMOS technology that frequently appear in ASICs hold at an intermediate voltage between logic 0 and logic 1, while other logic families can demonstrate oscillatory behavior [13].

These types of metastable behavior can be observed using analog circuit simulations, but are impossible to model in digital simulation due to the signal and device physics involved.

The best current method for handling metastability in digital simulations is to assign a random value to the flip-flop output following a setup time violation [14]. As demonstrated in Chapter 4, this basic model is supported by gate-level hardware simulators.

2.2 Embedded Software Fundamentals

2.2.1 Instruction Set Architecture

The instruction set architecture (ISA) of an embedded processor defines the set of instructions that the processor can perform. Instructions are the most basic unit of software execution and are designed to perform specific functions. Common examples found in most ISAs include `add`, `load`, `store`, and `jump` instructions. Instructions are encoded using specific formats that specify the type of operation, where the data comes from and where the result is stored. In general, instructions read and write data in the processor's register file, which is a collection of data storage elements. Beyond obtaining data from the registers, many instructions also support *immediate* fields, which are values hard-coded into the instruction format that can be used as input to processor operations. Despite having similar logical functionality, comparable instructions from different ISAs can differ greatly in their practical functionality.

First, ISAs often support multiple addressing modes, where the same register and immediate values can be combined in different ways to access memory. For example, the ARMv7-M ISA supports both pre-indexed and post-indexed addressing modes that allow a stored memory address to be automatically increased or decreased for iteratively accessing a block of memory [15]. The Atmel 8-bit ISA only supports pre-decrement and post-increment addressing modes, but not post-decrement and pre-increment modes [16]. The RISC-V 32-bit base integer ISA (RV32I) does not support any type of pre- or post-indexing mode [1].

Second, each ISA has its own instruction format that encodes the operation, data source and destination registers, and immediate values. For example, the instruction opcode and source/destination register fields in the RV32I ISA are in different locations compared to similar instructions in the ARMv7-M ISA. Even within one ISA, different addressing modes use the same bit locations in the instruction encoding in different ways. An example of this from RV32I is shown in Figure 2.1.

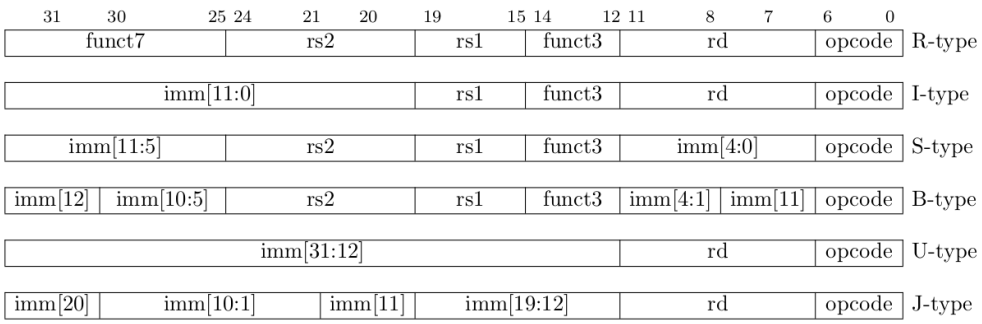


Figure 2.1: RISC-V 32-bit integer ISA instruction formats [1].

2.2.2 Processor Microarchitecture and Implementation

The ISA is only a specification of what computational functions an embedded processor can perform, and has few requirements for *how* processors are designed to accomplish them. While some ISAs define the number of cycles that each instruction takes, or expect certain memory hierarchy features, there are no requirements on how a digital circuit is designed to implement the processor.

The implementation of a processor can vary in two significant ways: in its microarchitecture and in its physical design. A processor's microarchitecture refers to the structure of components that make up the processor, including pipelining, memory hierarchy, and control structures. Beyond microarchitectural differences, two digital circuits that implement the same processor can be physically laid out on a chip in different ways, where different physical designs have different timing and parasitic characteristics.

Pipelining

The core unit of a processor is responsible for interpreting an instruction, fetching the necessary operands, performing any needed logic or arithmetic functions, and storing the result in the correct location. A simple single-cycle processor performs all of these actions within one clock cycle, by allowing the current instruction address to propagate through the datapath. However, most modern processors split instruction execution into a sequence of registered stages, called a *pipeline*.

In a pipelined datapath, the output of each functional step is stored in a pipeline register, allowing multiple instructions to be executed at the same time. In the example pipeline in Figure 2.2, instruction 0 will enter the pipeline in the fetch stage, and proceed to the decode stage on the next cycle. During the second cycle, instruction 1 will enter the pipeline in the fetch stage. The result of this clock cycle will have performed the fetch operation

on instruction 0 and the decode operation on instruction 1. This process continues, and, eventually, each pipeline stage is working on a different instruction.

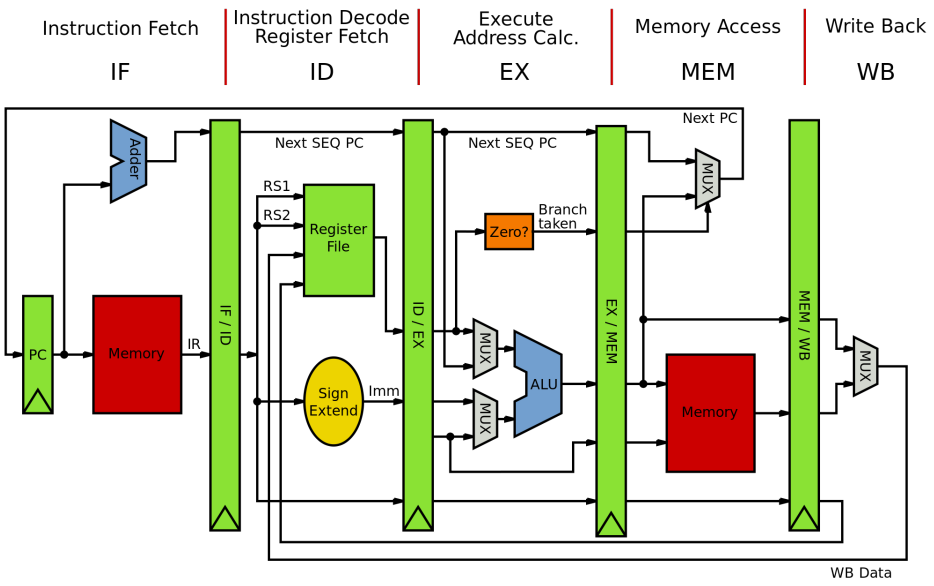


Figure 2.2: A pipelined MIPS processor.

Memory Hierarchy

While the processor core contains all functional units for processing an instruction and the entire processor register state, external memories are an integral part to any embedded computing system. The goal of the memory subsystem in any processor is to provide enough program memory and long term storage, while minimizing the time required to access a value. The most generic memory system may consist of a read-only memory (ROM) that stores the program code, and a random-access memory (RAM) for holding runtime variables. However, systems often have much more complicated memory hierarchies consisting of virtual addresses, translation lookaside buffers, caches, and memory management units. Complicated memory hierarchies with components like these vary from one device to another and greatly impact software execution.

2.3 Fault Injection Attacks

The basic idea of fault attack is that a system is presented with invalid inputs or environmental conditions that violate required operating parameters. The fault attack concept is applicable to any system, such as a factory machine that can only support certain power sources or loads. In these types of examples we think of an environmental violation less

as a fault attack, and more as a sabotage or blatant disregard of requirements. But just like mechanical systems, digital and embedded systems are susceptible to fault injection techniques that violate the environment operating parameters required for correct behavior. The fault injection methods discussed in this thesis are defined in the following section. The modeling study by Richter-Brockmann et al. provides further in-depth explanations of the device physics in these injection techniques [17]

2.3.1 Fault Injection Mechanisms

Clock Glitch Faults

Clock glitch attacks are achieved through modifying a system or local clock signal to violate timing requirements in the digital circuit. In such an attack, one or more periods of the clock signal are shortened to be less than the critical path delay defined in Equation 2.1. The width of the glitch pulse is chosen by the attacker, and can be made as short as desired. Clock glitches are a simple mechanism to inject faults, although they are less accurate than other methods. A modified clock signal affects all registers synchronized to it, so any register in the clock's domain is amenable to faults.

Voltage-Based Faults

Voltage-based attacks are achieved by either lowering the device supply voltage overall or causing a sudden significant drop in the supply voltage for a short amount of time. Both voltage underpowering and glitch attacks have been demonstrated to induce faults in the digital circuit state [17]. The mechanism by which voltage attacks result in circuit faults relates back to timing properties as well.

The time for NMOS and PMOS transistors to switch output values is inversely dependent on supply voltage. Therefore, a decrease in the device supply voltage results in an increased propagation delay for individual transistors in the circuitry. As a result, combinatorial gates together exhibit higher propagation delays for their digital values, which will likely extend beyond the width of the clock period. This effectively results in similar timing violations to those that occur in clock glitch attacks [17].

Electromagnetic Faults

Electromagnetic (EM) faults are performed by inducing current in the target device with an EM probe, which can lead to drops or overshoots in the voltage supply network. When an EM pulse forces a drop in the supply voltage, all signals currently at the positive supply

value drop to the negative supply value and only recover once the supply voltage drop is released by a second EM pulse.

In the case of a register near a clock edge, both the clock and data inputs may drop due to EM fault injection (EMFI). Upon returning to a regular state, the relative timing between when the clock and data signals recover is uncertain. If the rising edge of the clock signal recovers before the data signal, an incorrect value may be latched into the register [18]. In effect, EMFI creates sampling uncertainty, with potential violation of the register setup time. Compared to clock and voltage glitches, EMFI attacks are more precise in that a smaller subset of the device can be targeted. This gives the attacker more control over which bits in the device state may be affected by a fault attack.

2.3.2 Systemic Fault Injection Impacts

While fault injection techniques have been used to exploit both digital systems and embedded software [19, 20], a physical fault usually manifests as a 0 or 1 bit in the digital hardware state. The fault injection technique always causes a change at the hardware level, but whether faulty data will be stored in the registered hardware state is not always predictable. As shown in Section 2.4.1, there are few circuit fault evaluation methods that can distinguish a fault injection attempt from faulty bits that are induced in the hardware state. For the purposes of this thesis, the following definitions will apply:

- **Fault Injection / Attack** – The act of tampering with the system parameters/environment to cause faults in the hardware.
- **Fault Manifestation** – The process by which injected faults affect the circuitry and are either successfully incorporated into the hardware state or otherwise lost.
- **Hardware Faults / Faulty Bits** – The bits in the hardware state that are successfully affected by an injected fault.
- **Hardware Fault Propagation** – The process by which faulty bits propagate through the hardware, creating erroneous bits different parts of the circuitry.
- **Software Fault** – Bits in the software-level state that have been changed by hardware faults. Not every faulty bit in the hardware will have an impact on the correctness of software execution.
- **Software Fault Impact / Response** – The general changes in software behavior caused by software faults. This may be described quantitatively or qualitatively in terms of instruction execution.

For successful hardware faults, the following common models are used to describe fault impacts on state bits [17]:

- **Stuck At 0** – The hardware bit is forced to a 0 value
- **Stuck At 1** – The hardware bit is forced to a 1 value
- **Bit Flip** – The hardware bit is toggled from its current value

Even when a fault injection successfully induces faulty bits in the hardware state, there is a large abstraction stack that separates the hardware fault from software-level behavior. This long distance through technology physics, digital circuitry, microarchitecture, and ISA shown in Figure 2.3 is what makes design-time evaluation of software fault impacts challenging. Performing detailed analysis of physically-accurate hardware fault propagation makes it harder to track software running at the higher level of abstraction. Between different ISAs, microarchitectures, and physical implementations any two given embedded processors may respond differently to the same fault injection attacks. At the hardware level, differences in the physical circuitry and layout can lead to different impacts on the hardware state from the same fault. For example, the program counter in two different physical implementations of the same processor may produce different faulty states for the same clock glitch attack due to differences in the critical paths. At the architecture level, microarchitectural differences of the same ISA can lead to very different fault propagation behavior simply due to the nature of the microarchitecture having unique impacts on data flow.

2.4 Fault Vulnerability Evaluation Methods

The general goal of embedded software fault vulnerability evaluation is to know in which ways a given piece of software can be exploited by fault attacks. The vulnerability analysis can then be used to add countermeasures in software to protect the program, or in hardware to mitigate fault attacks in general. However, the heterogeneity of embedded processors in architecture, physical design, and ISA makes evaluation of processor fault vulnerability a hard problem to solve using any one solution.

Ideal vulnerability evaluation would cover as many fault injection techniques and parameters as possible, and be applicable to an entire software program while maintaining accuracy as to what is physically realistic for the target device. Unfortunately, these goals are not naturally compatible. Obtaining physically-accurate results requires device-specific details that are difficult to maintain at a higher level of abstraction, where full program analysis is much more feasible. As a result, current techniques in fault vulnerability evaluation are split

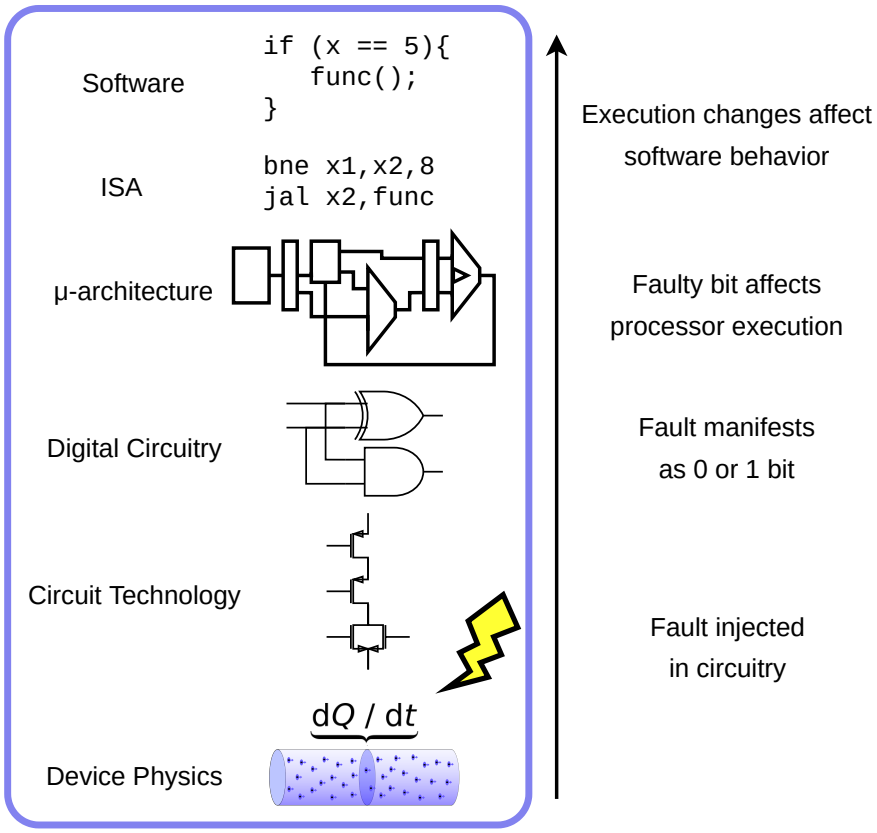


Figure 2.3: The impact of a fault injection attack through all of the main layers of hardware and software abstraction.

into four categories relating to one or both of the two goals of ideal vulnerability evaluation: physical accuracy and software coverage. The rest of this chapter presents publicly-available, state-of-the-art fault vulnerability evaluation solutions for these different focus areas.

2.4.1 Physical Hardware Fault Modeling

While the focus of this thesis is on software evaluation, hardware fault evaluation methods are of interest because they consider realistic fault manifestation and propagation. In pure hardware designs, fault attacks impact the internal state of the circuit and the faulty bits may propagate to the final outputs. Successful faults are determined by whether the fault propagates to the output, causing corrupted outputs that can then be used in differential fault analysis (DFA) attacks on cryptographic algorithms [19]. Understanding the underlying fault propagation is necessary for uncovering the root cause of software-level fault impacts.

Architecture Vulnerability Factor

The architecture vulnerability factor (AVF) is one of earliest methods introduced for evaluating the severity of a system's response to hardware faults [21]. AVF was created for non-adversarial contexts, considering only randomly occurring faults that are of interest for general fault tolerance testing. The method requires significant hand analysis, where the evaluator identifies all bits at the software level that affect the program output, called ACE bits. The AVF metric is then determined by calculating the amount of time the ACE bits spend in each hardware structure. Therefore, the AVF is unique to each hardware structure and is dependent on the particular software workload during analysis. Different software benchmarks result in different AVF values [21].

AVF is an example of heuristic fault vulnerability evaluation since specific ACE bits are identified based on an assumption that they are the only bits that can affect the output. Since this technique does not consider malicious faults, the final AVF is a risk metric that estimates an answer to the question: "How detrimental would it be for system operation if this hardware structure randomly had an error?" Furthermore, this method does not explore fault propagation or manifestation and instead treats all random faults as though they are guaranteed to cause failures. While this is an appropriate technique for estimating how critical each part of the hardware is for correct operation, it is not appropriate for malicious fault evaluation.

Timing Violation Vulnerability Factor

A more physically-accurate metric for fault vulnerability analysis is provided by the Timing Violation Vulnerability Factor (TVVF) [6]. TVVF is a metric for evaluating timing fault vulnerabilities in digital circuits using actual timing properties from the physical implementation. Determination of the TVVF for a circuit and set of faults is split into two phases: analyzing the scope of vulnerability (SoV), and analyzing the scope of propagation (SoP) [6].

Analyzing the SoV determines the likelihood that a given clock glitch results in a faulty register state. This is accomplished by performing static timing analysis on the combinatorial networks leading to register inputs and calculating the number of violated paths. This process considers the minimum clock period defined by the attacker model, and computes the probability of a successfully inducing faulty bits for different glitch widths starting with the critical path delay and decreasing down to the model minimum. Analyzing the SoP determines the likelihood that a successful glitch from the SoV propagates through downstream circuitry to reach a circuit output. This stage uses observability analysis, although the authors state that other techniques can be used as well [6].

TVVF was demonstrated on ripple-carry adder and AES S-box test circuits and is generally targeted for cryptographic hardware. While this technique was shown to be effective for

accurately determining hardware vulnerability to clock glitches, it is unknown how calculating TVVF would apply to processor hardware and how the computational intensity scales as the circuit grows. However, this is still an effective technique for achieving physically-realistic fault vulnerability analysis for critical pieces of hardware.

Verification Tool for Fault Injection

Arribas et al. introduced their verification tool for fault injection (VerFI), the first tool dedicated to evaluating adversarial fault vulnerabilities in hardware designs [5]. VerFI is a framework intended for testing hardware fault countermeasure coverage, and was designed to support user-specified fault models. The tool accepts an RTL description of the hardware and a fault configuration from the user, and automates the testing of multiple faults on a synthesized netlist.

VerFI synthesizes the RTL design into a netlist of logic cells, and builds a software representation of the circuit with additional properties at each node that allow for simulated fault injection. The fault configuration file controls the following properties of the injected faults: how many faulty bits to inject during the simulation; the maximum number of faults to inject during any clock cycle; the locations where the faults should be injected; the type of fault to inject. VerFI currently supports injecting faults at the inputs to submodules in the design hierarchy, and the authors state that in the future this could be extended to inject faults at the inputs of individual gates. The framework supports both bit flip and stuck-at fault models. VerFI also supports advanced groupings of target fault bits in the event that the user wants to evaluate the circuit's response to faults induced on an entire multi-bit register or bus.

These three main fault configuration features allow a range of faults to be simulated, including voltage and clock glitches, EM faults, and optical faults. Particularly, by injecting faults at the inputs to submodules, VerFI supports regional attacks where one part of a device is affected, but not another. This is necessary behavior for being able to cover EM faults. The simulation part of VerFI writes the fault configuration into the fault properties of the software copy of the netlist and uses event-based simulation to analyze fault propagation. Arribas et al. used VerFI to analyze the coverage of hardware designs protected with fault countermeasures and were able to find examples of faults that were properly mitigated, and faults that were not stopped by the countermeasures. One of the designs was a full protected AES accelerator, which demonstrates that VerFI is practical for use on non-trivial designs [5].

While VerFI is able to track fault propagation through a circuit, and allows the user to test resistance to a wide and flexible range of fault attacks, it does not inherently take into account which faults are more realistic to occur. Furthermore, VerFI starts with an RTL description of the hardware and runs synthesis as part of its flow. While this may be convenient in some situations, there may be designs that have specific physical layouts,

to achieve timing closure or insert fault countermeasures, for example. Therefore, it may be more practical for the user to provide a pre-synthesized netlist. This would also ensure that the fault simulation results are accurate with respect the final production version of the hardware, which may be different from the auto-synthesized netlist created in VerFI.

VerFI is an effective tool for exhaustively evaluating fault coverage of hardware countermeasures. Even without having knowledge of which faults are realistic, the ability to determine the number of logically possible faults that a countermeasure covers is valuable.

Adversarial Fault Modeling

Richter-Brockman et al. presented a method for modeling different types of fault injection mechanisms in terms of faulty bit behavior and circuit injection location [17]. Their adversarial fault injection modeling technique draws a distinction between a fault (attacker injection) and a fault event (the adoption of corrupted values into the circuit state). The modeling technique views the circuit as a directed graph of sequential and combinatorial elements, where a fault event may propagate through downstream nodes from the output of a sequential node. This view of the fault and circuit relationship is similar to the one used by TVVF, but allows for injection mechanisms beyond clock timing violations at the expense of losing physically-accurate fault manifestation information.

Modeling the actual fault event behavior is done in accordance to a list of fault manifestation models: Set/reset faults, bit flip faults, and a user-specified model for extension purposes. This allows any scenario that is achievable with real fault injection techniques to be represented by the model. For example, clock glitches can be modeled by placing a set/reset fault type on all combinatorial gates, since the faulty bits latched into the state come directly from early gate outputs. EM faults are modeled using set/reset faults as well, but with faults applied to sequential gates since EM pulses cause sampling errors [17]. The study also defines a "fault coverage" metric that measures how many fault events propagate to an output across the combination of all attacker faults and circuit inputs. They provide a trivial example of a XOR-with-parallel-inverter circuit that has been duplicated as a countermeasure. The metric determined that the fault coverage of the example circuit was 93.75%, meaning the duplication countermeasure prevented 93% of faults from propagating to the output.

The authors implemented their model using VerFI, where using a specific fault injection mechanism limits the number of fault configuration parameters. Since the adversarial model supports application of faults to combinatorial gates, sequential gates, or both, the full model may not yet be supported by VerFI. While this more formal look at fault modeling provides guidelines for how to represent the manifestation of different injection techniques, it still does not take into account the device physics. Therefore, this model still involves testing all logically possible hardware faults and does not evaluate which faults could realistically

occur. For example, in an EM fault there is non-deterministic sampling of register data inputs since the timing with which the data and clock signals recover is unknown. However, being able to restrict the total number of possible faults with a model like this is beneficial, as it allows evaluation of specific types of fault injection techniques.

2.4.2 Physical Software Fault Testing

While this thesis explores capabilities for evaluating fault vulnerability before obtaining a physical device, knowledge can still be gained from from studies on real physical attacks. Particularly, the results from physical attacks inform what types of behavior should be achievable from a physically-accurate, pre-production software fault vulnerability framework.

Fault Characterization

In the past decade, multiple studies have focused on characterizing the effects of embedded software fault attacks on various platforms. While the results vary by platform, each study confirms that most results can be explained by replacing the original instruction with a different one [22] [23, 24]. Each of the following studies aimed to describe how EM faults on a given target platform are likely to impact the program state and outputs.

A 2013 study by Moro et al. tested EM faults on an ARM Cortex-M3, focusing heavily on applying faults to a single load instruction while fetching it from flash memory [22]. The authors concluded that all faults that are explained by single-instruction-replacement must be faults on the instruction fetch, and that all unexplained outcomes are faults on the data fetch. These observations culminate in a simple model where the time of the fault corresponds to the number of 1s that will be read from the memory data line. This result was deemed device-dependent because the bus precharge value is responsible for whether a stuck at 0 or stuck at 1 fault manifests.

A study by Proy et al. in 2016 provided more insight into fault behavior on an ARM Cortex-A9, compared to the single instruction analysis from the prior study [23]. The first set of experiments aimed only to determine how the final results of a simple data-copying loop changed in response to various EM faults. The results were split into 4 levels of severity: no effect, program crash, output corruption, program flow corruption. A program flow corruption indicated that both faulty outputs were observed and the loop took an incorrect number of cycles to complete [23]. These types of outcomes relate back to the general goals of cryptographic fault attacks, where the attacker wants to obtain faulty outputs while maintaining correct program flow.

The second set of experiments focused on different simple instruction sequences to build up a set of heuristics describing how single faults can affect instruction execution. The following is a list of observable behavior that explained most faulted instruction outputs [23]:

- Instruction Skip (includes instruction replacement via a NOP)
- Register upper half-word reset
- Full register corruption
- Source operand substitution
- Instruction replay
- Repeated fault effects

Beyond characterizing effects on simple instruction sequences, Proy et al. also applied this type of characterization to multiple versions of the data-copying loops with various countermeasures applied. The results showed that traditional software-level countermeasures may work against simple instruction skip models, but are ineffective against more complex fault effects [23]. Similar studies were performed by Trouchkine on multiple embedded platforms, finding similar instruction level replacements and effects [24]. Beyond explaining fault responses with an ISA-level model, Trouchkine designed specific instruction test sequences to exercise different parts of the microarchitecture. For example, a memory store and load test was able to distinguish between the caches and memory management unit being faulted during the attack.

Attacks Exploiting the Microarchitecture

The previous discussion focused on using EM fault attacks to characterize how different faults affect instruction execution using ISA-level and microarchitectural explanations. However, malicious attacks that use microarchitectural information to their advantage also demonstrate how device specifics have a great impact on fault attack outcomes. Yuce et al. demonstrated successful single clock glitch attacks against instruction level countermeasures in the LED block cipher [25]. These attacks leveraged the fact that the target device used a 7-stage pipeline to determine when a fault could be placed to bypass duplication, triplication, and parity countermeasures.

Furthermore, the attacks considered when different microarchitectural blocks would be in use. Even without knowing the exact physical layout and timing of the circuit components, sufficient assumptions could be made about which pipeline stages would have the shortest path. For example, a branch currently in the execute stage would not require the ALU,

resulting in a short critical path. Therefore, faults could be injected on instructions in other stages without interfering with the execute stage [25].

These observations on the impact of both microarchitecture and circuit parameters on fault attacks are critical for evaluating software fault vulnerability in the future. Effective fault vulnerability evaluation techniques should consider both the explainable results found in the characterization studies, as well as these hardware-specific effects that enable successful attacks on protected software.

2.4.3 ISA-Level Fault Vulnerability Simulation

A study by Yuce et al. investigated using fault responses of embedded processor instructions to improve the success of fault injection attacks [26]. Microarchitecture-Aware Fault Injection Attacks (MAFIA) were accomplished by first profiling the target device in simulation to learn how clock glitch attacks affect the execution of different instructions. The resulting information was then used to craft clock glitch fault attacks that target specific instructions and minimize the effect on other instructions in the processor pipeline.

To determine the fault sensitivity of different instructions in each pipeline stage, the first phase of MAFIA involves running gate-level timing simulations of the target platform as it executes the target instruction. When the instruction is in the target stage, a clock glitch is injected through the simulation testbench and the instruction output is compared to the expected value. This process is repeated with increasingly shorter clock glitch widths until a faulty value is observed in the instruction output. While this technique is effective for simulating realistic fault manifestation of the clock glitch mechanism, the only metric measured is the exact glitch width that causes the first output corruption. Furthermore, the simulation method for MAFIA is only applied in the context of building more powerful fault attacks. The results of profiling only explain which faults cause errors in the software, but not how the software or hardware is affected by fault attacks. However, a similar simulated injection mechanism is used as the basis for the SimpliFI framework presented in Chapter 3 and is shown to be an effective technique for capturing both hardware and software fault propagation in Chapter 5.

2.4.4 Software Fault Simulation

The fault evaluation methods discussed so far have either applied to hardware accelerators, or involve testing a real physical device. Physical device testing is effective for characterizing an existing, produced device, but has challenges of instrumentation and restricted data

collection. Hardware-level evaluation and modeling techniques are effective for tracking fault propagation through circuitry, but have limited context of what happens at the software level. When analyzing software-level fault vulnerabilities, the attack surface expands; output data corruption may only be one attack vector that can give the adversary a favorable outcome. For example, an attack that induces a fault in the program and bypasses cryptographic operations entirely may require less work on the adversary's part, relative to performing DFA on faulty outputs. These potential "bypass" attacks have been known since the early days of fault injection, particularly in the form of faults that force a processor to skip an instruction [20]. Software-level fault impacts such as this are easiest to identify and track with ISA-level simulation.

One ISA simulation tool designed to evaluate software fault attack vulnerabilities is Riscure FiSim [7]. FiSim currently supports evaluating software for ARM architectures, allowing the user to input a platform model defining the address space, memory regions, and stack information. The tool runs an ISA-level simulation of the program using this information, and allows emulates fault injections on arbitrary instructions using either an instruction skip or instruction encoding bit flip model. However, users are able to add their own software-level fault models to the simulator.

This type of fault evaluation is important since it can exhaustively evaluate fault propagation through software in response to different instruction-level fault models at different points in the program. This is much easier than having to instrument a device and track the program's progress to correctly inject the fault, as is necessary with physical testing. However, a significant downside to ISA-level simulation is the lack of hardware-specific results, both in terms of microarchitectural effects and realistic fault manifestation. Still, in situations where common faults, such as instruction skips, need to be evaluated, ISA simulation provides an easy method for performing a high-level first pass.

2.5 Discussion of State-of-the-Art

The previous section introduced four methods for evaluating fault attack vulnerability. First, physical modeling techniques such as TVVF and VerFI provide ways to evaluate fault propagation and manifestation through hardware. TVVF supports both realistic fault manifestation analysis and fault propagation through downstream circuitry [6]. VerFI supports arbitrary fault propagation simulation through hardware accelerators with a focus on evaluating coverage of hardware fault countermeasures [5]. By itself, VerFI does not incorporate hardware-specific fault models, but an adversarial fault model can be used to restrict the fault space to represent faults that are logically possible with a specific injection technique [17].

Next, early fault injection attacks on embedded software established that instructions can be skipped, enabling both cryptographic attacks (DFA) and control flow attacks on embedded software [20, 19]. Next generation analysis extended this understanding to include single instruction replacement [22]. Since then, the instruction replacement model has expanded to include instruction replays, register data corruption, and operand substitution [23]. Furthermore, using these instruction-level observations in conjunction with carefully-crafted test sequences enables some level of microarchitectural analysis through physical fault testing, without detailed knowledge of the microarchitecture [24]. However, simple glitch attacks that leverage knowledge of the microarchitecture to bypass software countermeasures show that having knowledge of the hardware greatly improves insight into fault vulnerability [25].

Third, attacks that use knowledge of microarchitecture were improved further by profiling fault sensitivity of different instructions on the target device. This was achieved by using timing simulations to detect the first clock glitch width that causes instruction output corruption at various instruction stages [26]. However, this type of analysis was only used to improve the accuracy of attacks and only determines which faults affect instructions and not *how* they affect the instruction execution. Finally, ISA-level simulators that support arbitrary fault injection provide rapid evaluation of software fault vulnerability. Despite the ability to cover a wide range of faults and simulate simple fault models, ISA simulation does not inherently support targeted testing of realistic fault events.

VerFI, and ISA-level simulators like FiSim seem to achieve similar capabilities. They both simulate fault propagation in their respective domains (hardware and software) and support arbitrary fault injection, regardless of whether the faults are realistic. However, FiSim is targeted towards single instruction faulting, and does not consider how the microarchitecture may cause multiple instructions to be faulted at once. The observations made by Yuce et al. during the single glitch attack study hints that this type of modeling may be possible in FiSim. The challenge would be adding microarchitectural awareness that correctly handles complex situations such as out-of-order and speculative execution.

A summary of the benefits and tradeoffs between these types of fault evaluation methods is given in Table 2.1. Physical device testing covers all functional categories, but due to the challenge of instrumentation, only limited information is available on hardware and software internals. What is currently missing from this collection is a pre-production method that can consider physical manifestation like TVVF, physical device testing, and MAFIA, complex circuit propagation like VerFI, and program-level analysis like FiSim.

Table 2.1: Summary of fault evaluation method capabilities.

Method	**SW Analysis**	**SW Propagation**	**HW Analysis**	**HW Propagation**	**Fault Manifestation**	**Injection Methods**
TVVF	○	○	●	●	●	◎
VerFI+Modeling	○	○	●	●	○	●
MAFIA	◎	○	◎	◎	●	◎
Device Testing	●	◎	◎	◎	●	●
FiSim	●	●	○	○	○	○

○ = No Support ◎ = Limited Support ● = Full Support

Chapter 3

Fault Attack Simulation Framework

The current fault evaluation methodologies summarized in Section 2.5 each highlight the important aspects of effective fault injection analysis. This chapter presents the SImulation Methodology for embedded Processors to Learn the Impacts of Fault Injection (SimpliFI), a general framework that supports the best capabilities from other current methods. By combining physically-accurate fault manifestation and hardware-level propagation analysis with a focus on software-level evaluation, SimpliFI is the first methodology for realistic pre-production software fault vulnerability analysis. Before the implementation described in Chapter 4, SimpliFI is first introduced as a collection of design principles, functional requirements, and additional features that can be applied to different tools and platforms.

3.1 Design Principles

SimpliFI is defined by tool- and device-agnostic requirements to ensure that implementations which follow the guidelines achieve all of the primary features of current evaluation methods. Furthermore, the framework describes only the necessary features of an implementation, leaving room for extra analysis features that may be helpful for a given set of tools and devices.

Simulate Realistic Fault Manifestation Using a post-layout netlist makes physical circuit properties available for simulating fault manifestation. For example, approximate power-consumption and signal timing can be obtained from an SDF file that is unique to the processor implementation. While VerFI uses a synthesized netlist of device components, the benefit of having hardware-level information is lost by using a software representation of

the circuit. A SimpliFI implementation should use a simulation method that can leverage physical circuit properties to emulate different fault injection techniques.

Capture Hardware Fault Propagation In order to determine how the simulated faults impact the processor state, hardware-level signals should be tracked so that faulty software-level outcomes can be traced back to corrupted hardware state bits. While physical circuit properties are already required by the fault manifestation design principle, this fault propagation principle requires that the hardware state be actively tracked during execution, and not just during fault injection.

Support Software-Level Analysis The final results should be tailored towards evaluating software-level behavior. Therefore, SimpliFI implementations must be able to collect software-relevant state at the end of a test, including processor registers, the program counter, and the final processor hardware state. These results should contain at least as much information than was shown being collected during physical testing [23, 24]. Furthermore, in conjunction with tracking the hardware state through execution, software-level data collected by SimpliFI gives users more information than is possible with physical fault testing methods, where the microarchitectural state is inaccessible.

3.2 Framework Design and Features

Figure 3.1 depicts the full SimpliFI framework, supporting the design principles described in the previous section. The gate-level hardware simulation core achieves hardware-specific results, while the outer layer supports software-level analysis and test management. SimpliFI mimics the nature of the hardware/software relationship; in reality, software is just an advanced configuration of the hardware, and the hardware is the entity that actually does computational work.

The inner and outer layers of the framework can be implemented to support analysis of both simple instruction sequences and full programs. Testing short instruction sequences can aid the user in evaluating general fault vulnerabilities in the embedded processor. This is akin to studies that use physical testing to characterize a device's fault response behavior [24], except SimpliFI can also detect how faulty bits propagate through the hardware. Testing full programs aids the user in evaluating realistic vulnerabilities in critical security software. Section 3.2.1 discusses the benefits of full program analysis, and Chapter 6 discusses how processor characterization with SimpliFI can potentially build a model for use in ISA-level simulation. For the purposes of this thesis, SimpliFI is only tested and implemented for

evaluating clock glitch faults. However, Section 3.2.2 discusses how additional injection techniques can be modeled in the same framework.

To clarify the difference between user-designed tests and fault injection attacks simulated in the framework, the following terminology applies:

- **Test** – A user-defined program and configuration pair that instructs the simulator to apply faults at different points in the program. A test may involve multiple subtests.
- **Subtest** – A subtest is one part of a larger test case which specifies a start point, target point, and multiple fault injection trial parameters.
- **Fault Injection Trial** – A singular execution of the test program with one fault applied at a specific point.

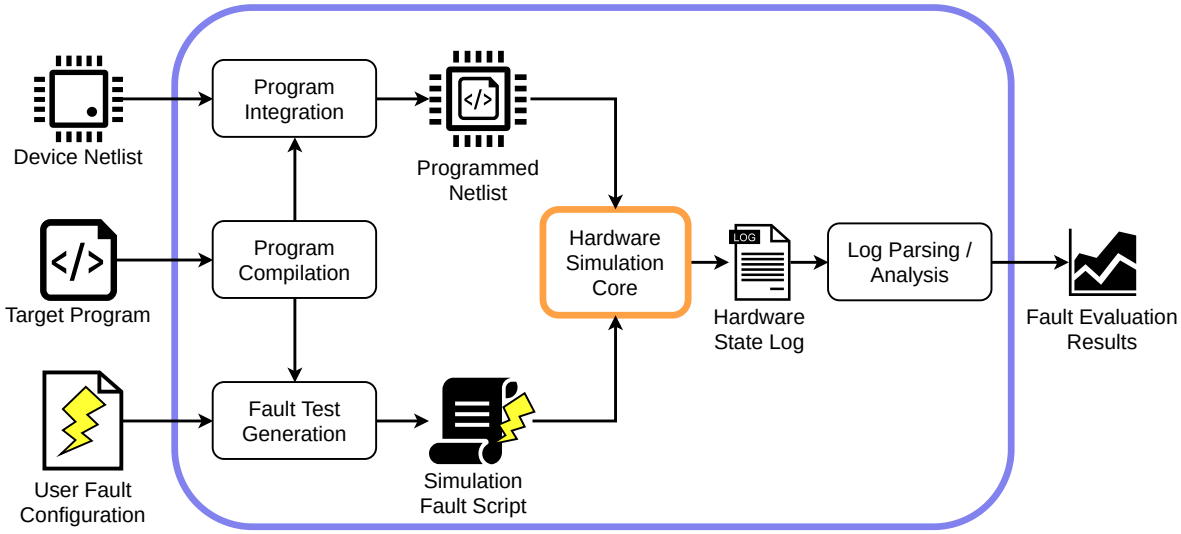


Figure 3.1: High-level depiction of the SimpliFI framework.

3.2.1 Outer Framework: Software-Centric Control

The outer layer of SimpliFI is responsible for building the device simulation environment for a specific program, generating a complete set of fault test cases, and analyzing data collected during simulation. The following sections discuss the main phases of the outer framework, and define the functional requirements a SimpliFI implementation must achieve.

Program Compilation

The first step of the outer framework compiles user test programs into binaries that can run on the embedded processor. To support both sequence and application testing, SimpliFI accepts either a main assembly file or main C program file. The exact compilation process is highly device- and program-dependent, so the framework does not have further restrictions other than requiring a binary.

Program Integration

The purpose of the program integration step is to optionally create a copy of the device netlist with the program included. Since many embedded processors rely on external memories to store the program, this may not be a practical step for some platforms. However, for any platform where the program cannot be included in the netlist, the program must be fed into the processor as part of the simulation testbench (Section 3.2.2).

An example where the integration step is necessary is when an FPGA-based processor stores a program in an internal memory primitive, such as Xilinx Block RAM [27]. In this case, the program can be stored in component configuration parameters, as the implementation in Chapter 4 does. Regardless of whether an explicit program integration step is used or not, the main functional goal that must be supported by an implementation is making the embedded processor run the target program during simulation.

Fault Test Generation

One of the key features of SimpliFI is its ability to automate evaluation of multiple test cases with a range of fault injection trials. The outer layer of the framework creates a fault simulation script for the simulation core which specifies all of the fault injection trials that will run. The simulation core expects a full specification of all parameters for each subtest:

- Program Type
- Start Point
- Target
- Observe Point
- Starting Glitch Period
- Ending Glitch Period
- Glitch Period Step

The three parameters related to the clock glitch width hold the same meaning for instruction and application tests, while the other parameters vary in purpose for the two test types. In an instruction test, the *start point* parameter indicates the memory address of the instruction being evaluated, and the *target* parameter specifies the instruction execution cycle that should be faulted. This allows the user to create multiple subtests to test the fault response of different processor pipeline stages. In these tests, the observation point specifies the number of clock cycles after the start point when the instruction output should be recorded. For most instructions, this is the number of clock cycles required to process the instruction. However, since the full device memory is not recorded by the simulation core, the observation point could be set to a later time in the case of memory store instructions, where the value written into the memory can be read back into a processor register for observation.

Since applications have significantly more complicated program flow than a short sequence of instructions, the address of an instruction is not a unique-enough identifier for when the simulator should inject a fault. Instead, the test program can be instrumented using a unique macro or function that is called when the simulator should prepare to inject the fault. In this case, the *start point* parameter identifies the memory address of the macro, and the *target* identifies where the target instruction is following the macro. The observation point is determined in a similar method to the start point. An example of this style of instrumentation is provided in Chapter 4.

To simplify user configuration of the simulation tool, SimpliFI defines a custom and flexible file format for the user to specify test configurations. Instead of fully specifying all parameters for every subtest, the user can set global parameters that apply to every subtest, and then create shorthand entries for different instruction and cycle subtests. Before SimpliFI starts up the simulation core, it converts the user configuration file into a fully-specified fault simulation script that the simulation controller can understand. An example user configuration file and its corresponding fault simulation script are shown in Listings 3.1 and 3.2, respectively. In the example user configuration file, the *GStep*, *GStart*, *GEnd*, *Observe Point*, and test type are set for all following subtests. With the "@@" characters as subtest delimiters, this configuration file will create subtests for the instruction at address 2C for stages 0 through 7, and the same for the instruction at address 4C.

Listing 3.1: Sample user fault configuration file.

```
SEQ

GStep: 0.5
GStart: 12
GEnd: 4
ObservePoint: 7

StartPoint: 2c
Target: 0,1,2,3,4,5,6
@@

StartPoint: 4c
Target: 0,1,2,3,4,5,6
@@
```

Listing 3.2: Sample simulation fault script produced by the test generation step.

```
SEQ
StartPoint: 2c
Target: 0,1,2,3,4,5,6
ObservePoint: 7
GStep: 0.5
GStart: 12.0
GEnd: 4.0
@@

SEQ
StartPoint: 4c
Target: 0,1,2,3,4,5,6
ObservePoint: 7
GStep: 0.5
GStart: 12.0
GEnd: 4.0
@@
```

Output Processing

SimpliFI is able to analyze both hardware fault propagation and software-level outcomes by leveraging the hardware state information recorded by the simulation core. The post-processing performed on the data supports the same types of analyses as physical device testing, where program and instruction outputs are inspected for errors. The output processing stage handles analysis of instruction sequence and full application tests differently.

For instruction sequences, the goal of post-processing is to determine how the instruction output and hardware state are impacted by different fault parameters, with faults being injected at different stages across multiple subtests. SimpliFI computes the hamming distance (HD) between actual execution outputs, and the expected values observed during a clean run of the same program. The HD analysis identifies all registers that were corrupted in at least one subtest and calculates how the final value is affected by each fault injection trial.

The goal of post-processing for full program tests is to determine which faults affected the final program outputs, which faults crashed the program, and which faults had no effect on the final output. The same final output data is collected as in the instruction tests, but with a focus on program-level behavior. Since the instruction tests are intended to explore the fault responses of microarchitectural components and only run for a few cycles, processor failures are unlikely. On the other hand, full applications may run for hundreds or thousands of

clock cycles, so analysis focuses more on program-level and less on the hardware. Chapter 5 discusses how tests can be written to evaluate specific aspects of the hardware or software.

Although the focus is on program-level effects, SimpliFI can analyze how fault propagation through the hardware state corresponds to different program outcomes. While some software-level corruption may primarily be the outcome of software-level error propagation, it is possible that some software outcomes consistently correspond to the same types of hardware-level fault propagation. These main analysis features are only a starting point for the output processing stage, and users can implement their own metrics and analytic methods to extend the results of the SimpliFI framework.

3.2.2 Inner Framework: Hardware Simulation Core

The inner layer of the SimpliFI framework is responsible for injecting faults into the simulated hardware and tracking the hardware state immediately after the fault and at the end of the test. A functional diagram of the simulation core is shown in Figure 3.2.

Simulation Functionality

SimpliFI uses post-layout gate-level timing simulation, which automatically incorporates timing properties of the entire netlist into simulated fault manifestation. There are numerous hardware simulation tools that support physical netlist timing simulations, which are developed to efficiently evaluate hardware [28, 29]. While it is possible to write a custom simulator that has built-in fault support, such as VerFI, simulating with a dedicated hardware tool is likely faster and more efficient [5].

The simulation core control level is responsible for reading each subtest configuration, such as the ones in Listing 3.2, and loading the parameters into the hardware testbench. In addition to the fault injection trials, the controller runs an initial clean test for each configuration to gather the expected values and runtime. This baseline data is necessary for effective fault response analysis during output processing.

The testbench functionality shown in Figure 3.2 can be achieved with a SystemVerilog module. SystemVerilog supports high level modules, such as a testbench, reading values from lower levels of the simulated hardware. This feature allows the internal state to be recorded at arbitrary time points during simulation. Additionally, hardware simulators allow the simulation control script to write/read values in the testbench [28, 29]. This is one mechanism for loading the test parameters into the testbench, and is the method used in the implementation in Chapter 4.

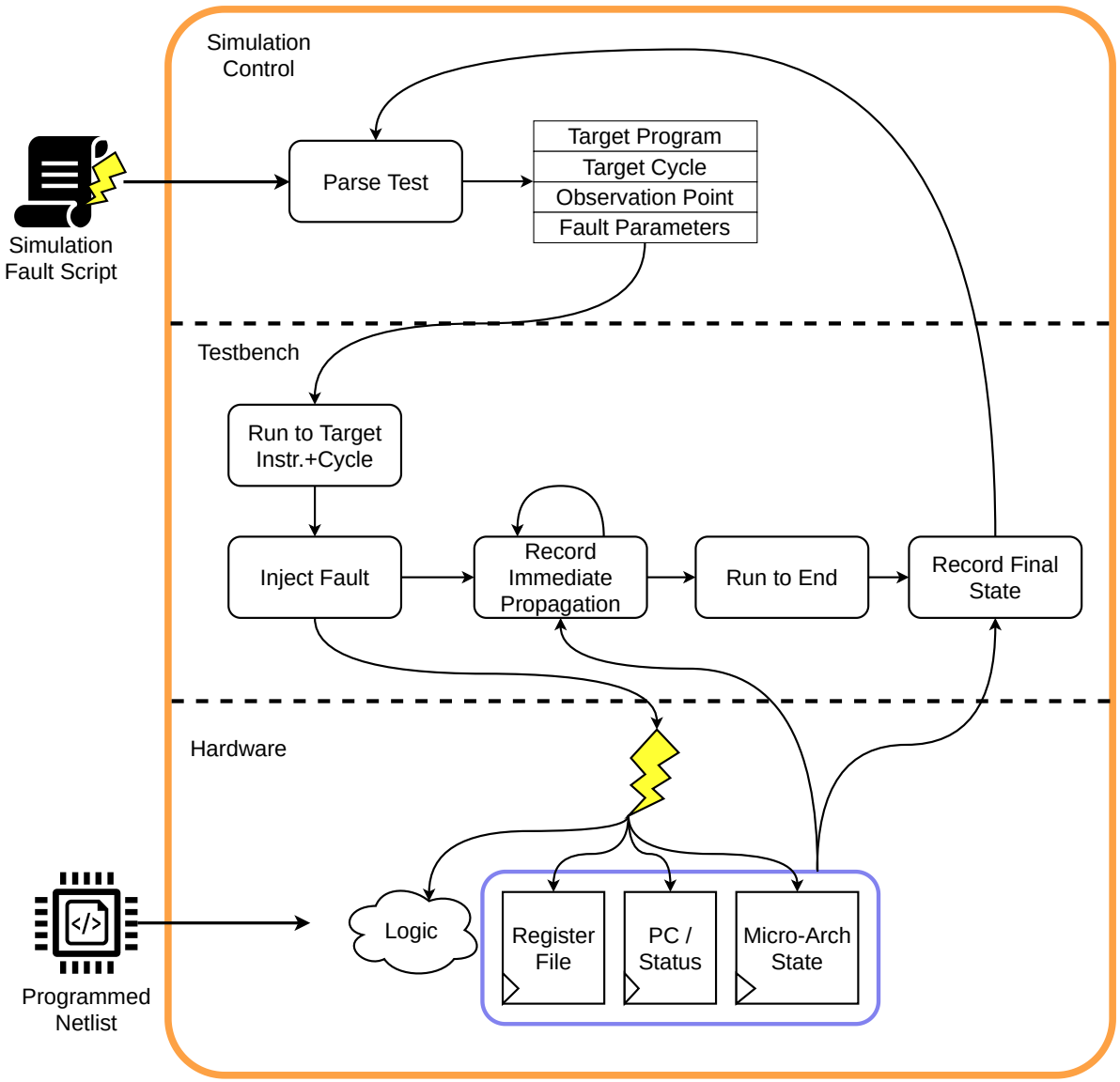
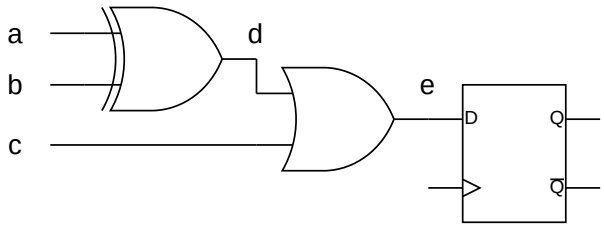


Figure 3.2: SimpliFI hardware simulation core diagram.

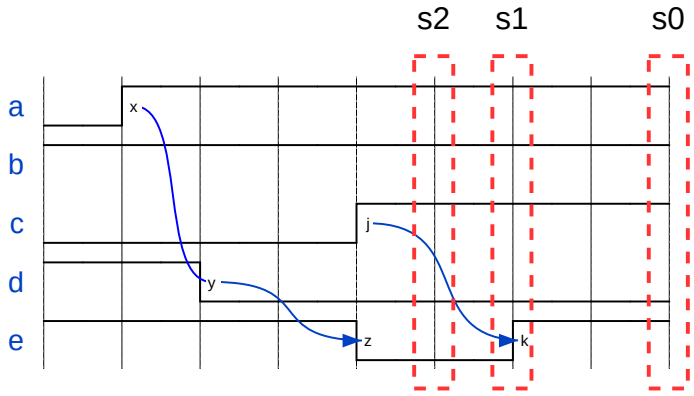
Clock Glitch Injection Mechanism

A critical part of SimpliFI is its ability to inject realistic faults into the hardware. As stated earlier, this thesis focuses on simulating clock glitch fault attacks, although other injection mechanisms can be simulated as well. By using post-layout gate-level simulation, the timing information required to emulate clock glitch attacks is automatically included in the simulated netlist behavior. During timing simulation, gate outputs are updated according to their propagation delays defined in an SDF file.

With signal propagation time enforced by the simulator, the testbench can emulate a clock glitch by manually shortening the clock period for one cycle and then returning to the correct clock frequency. If the period is not short enough to violate signal timing, no faulty bits are latched into the registers and the simulation will continue to run as if no fault was injected. However, if the clock period does cause timing violations, then faulty bits may manifest in the hardware state and propagate through the program. While any violation of the critical path constitutes a successful clock glitch event, the resulting faulty bits are caused by two distinct timing events.



(a) Example circuit.



(b) Signal propagation with sampling windows covering the register setup time.

Figure 3.3: Timing violations at different sampling points.

To demonstrate the two event types, consider the circuit shown in Figure 3.3a. The `XOR` and `OR` gates both have propagation delays from input to output, which for the purposes of this example are just considered to be greater than 0 nanoseconds. The timing diagram in Figure 3.3b shows signal propagation in response to the input $\{a, b, c\}$ changing from $\{0, 1, 0\}$ to $\{1, 1, 1\}$, with `a` transitioning before `c`. If the clock edge occurs at the end of the `s0` window in Figure 3.3b, the correct `e` value is latched into the register. If the clock edge occurs at the end of the `s1` window, a setup time violation occurs due to `e` transitioning during the setup windows. In this case, the register state is unpredictable and may even become metastable. Finally, if the clock edge occurs in `s2`, there is no setup time violation since the data does not change in the sampling window. However, the temporary 0 value on

e will be latched into the register. While both of these events count as timing violations, one of them violates the setup time, and the other causes an early incorrect sample.

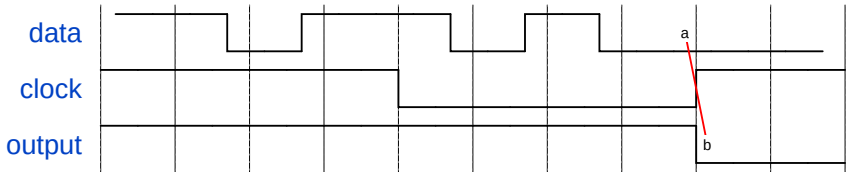
Hardware simulators naturally handle incorrect sampling events, since the propagation delays are modeled correctly so that signal e appears as a 0 at sampling point. However, the setup time violation event is more complicated since setup time violations in a real circuit can lead to register metastability. As discussed in Section 2.1.2, current metastability models rely on analog characteristics of the register and data signal to characterize metastable flip-flop outputs with some level of accuracy. Since digital gate-level simulation abstracts away the underlying device physics, there is not enough information to simulate potential metastable behavior using the existing models.

However, SimpliFI supports a random metastability model as a best effort to simulate setup time violations. Instead of latching the data signal value at the exact time of the clock edge, a random value is assigned to the register state. While metastability is not a fully random phenomena, this technique acknowledges that unpredictable values may be introduced into the hardware state as a result of clock glitches that cause setup time violations.

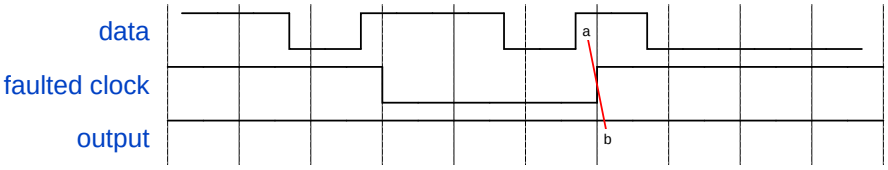
Injection Mechanism Extensions

The underlying physical effect SimpliFI leverages for fault injection is timing violations, with the injection technique being a clock glitch. Therefore, other timing-based faults could be added to the framework for future extensions, including voltage-based faults and even EM faults. As discussed in Section 2.3.1, voltage-based faults also disrupt circuit timing to inject errors into the state. Figure 3.4 shows example clock and voltage glitch fault effects on a register. A voltage glitch attack increases the propagation delay of data signals, leading to longer critical paths. Figure 3.4c shows this, with the normal data transitions from the clean sample taking a longer amount of time to update in the faulted version. In this example, the clock and voltage glitches can lead to setup time violations or critical path violations. In both cases, the rising clock edge occurs closer to the data transition times; the clock glitch moves the clock edge closer to the data, while the voltage glitch moves data transitions closer to the clock edge.

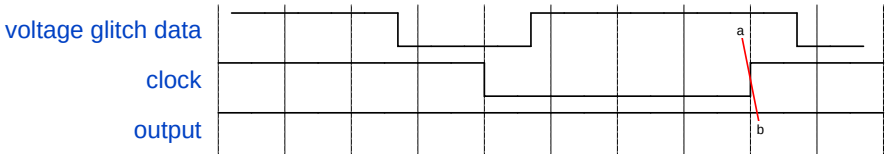
These shared properties are one potential way to achieve voltage glitch simulation in the SimpliFI framework. The clock glitch mechanism already moves the clock edge closer to the data transitions, and could be used in the same way to simulate voltage glitches. The key challenge that needs to be addressed is calculating how the clock glitch width maps to a particular voltage glitch. More work would be required to determine this relationship, but this is a potential starting point for supporting more injection techniques. Voltage underpowering attacks have similar effects as voltage glitch attacks, so the mechanism for voltage glitches could be applied for a longer period of time to simulate voltage underpowering. While EM faults are more complicated than clock and voltage attacks, the underlying fault manifesta-



(a) Clean data sampling.



(b) Data sampling with clock glitch



(c) Data sampling with voltage glitch

Figure 3.4: Clock and voltage glitch timing effects.

tion is caused by setup time violations with data signals at the positive supply voltage [18]. Since SimpliFI already supports basic metastability modeling in setup violations, EM faults could be integrated into the framework by modifying data signals in the device that would realistically be affected by EM pulses. This is one extension for future work on SimpliFI that would greatly increase the versatility of the framework beyond its current abilities.

3.3 Summary

This section introduced the SimpliFI framework for evaluating software fault vulnerabilities. SimpliFI simulates realistic fault manifestation by using gate-level simulation and injecting faults that cause timing violations. The outer layer of the framework facilitates test control and separates the user from the lower level simulation core while still giving them control over program targets and fault injection parameters. The framework supports both short instruction sequence and full application testing, with different analysis techniques applied to each evaluation type. The low-level simulation core handles fault injection and hardware state tracking. By recording the hardware state, SimpliFI can track fault propagation through the hardware during execution, while still supporting software level analysis to determine how faults affect software execution.

Chapter 4

BRISC-V Framework Implementation

The SimpliFI framework outlined in Chapter 3 was implemented for the BRISC-V platform created by the Boston University Adaptive and Secure Computing Systems Lab [8, 9, 10]. This Chapter discusses how each of the framework components was implemented to support automated testing and analysis of instruction sequences and applications. This implementation of SimpliFI targets a Xilinx FPGA implementation of BRISC-V processors, so the code and tool features are vendor-dependent. However, other hardware design tools such as ModelSim, Cadence, and Synopsys support similar features.

4.1 BRISC-V Platform Overview

BRISC-V is an open-source RISC-V processor platform hosted by Boston University that allows users to customize a processor implementation with different pipeline lengths, memory hierarchies, and memory sizes. The customization used in this thesis had a 7-stage pipeline and a single memory that stores both the program code and data. Figure 4.1 depicts the selected processor pipeline.

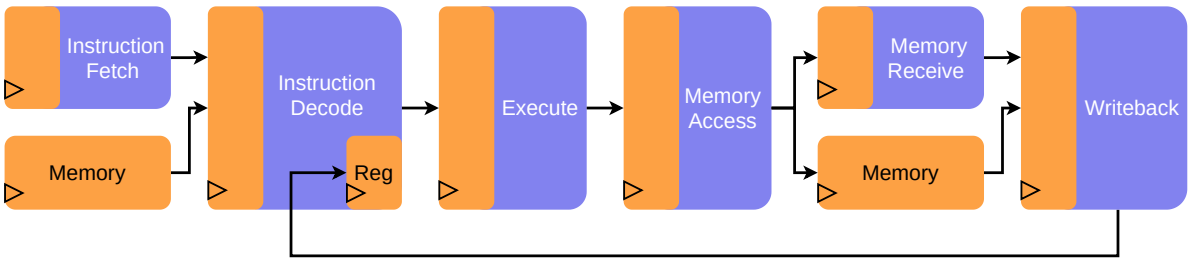


Figure 4.1: BRISC-V 7-stage pipeline.

When implementing the BRISC-V processor in Xilinx Vivado, the original netlist did not utilize Block RAM due to an incompatible memory access process in the Verilog code. To bypass this and achieve a netlist that uses Block RAM for memory, the original memory access code (Listing 4.1) was modified (Listing 4.2). The main functional difference between these two versions is that when the memory is written to, the write value propagates to the output bus. This did not cause any execution errors when testing large applications to verify validity. Otherwise, the changes in enable logic preserve the original enable logic, but with only one `if` statement per port as is required by Vivado Block RAM inference.

Listing 4.1: Original BRISC-V memory access code.

```
1  // Port 1
2  always@(posedge clock) begin
3    if(writeEnable_1)
4      // Blocking Write to read new data on read during write
5      ram[address_1] = writeData_1;
6    if(readEnable_1)
7      readData_1 <= ram[address_1];
8  end
9
10 // port 2
11 always@(posedge clock)begin
12   if(valid_writeEnable_2)
13     // Blocking Write to read new data on read during write
14     ram[address_2] = writeData_2;
15   if(readEnable_2)
16     readData_2 <= ram[address_2];
17 end
```

Listing 4.2: Modified BRISC-V memory access code to support Vivado Block RAM inference.

```
wire en1;
wire en2;
assign en1 = writeEnable_1 | readEnable_1;
assign en2 = valid_writeEnable_2 | readEnable_2;

// Write before read Vivado synthesis model
// Port 1
always@(posedge clock) begin
  if(en1)
    if (writeEnable_1) begin
      ram[address_1] <= writeData_1;
      readData_1 <= writeData_1;
    end else
      readData_1 <= ram[address_1];
end
// port 2
always@(posedge clock)begin
  if(en2)
    if (valid_writeEnable_2) begin
      ram[address_2] <= writeData_2;
      readData_2 <= writeData_2;
    end else
      readData_2 <= ram[address_2];
end
```

4.2 SimpliFI Implementation

This section discusses the tool-specific aspects of implementing SimpliFI for BRISC-V, as well as some of the high-level automation for running tests. The tool was implemented using a combination of Python3, TCL, and shell scripts. A brief overview of each file is given below, and with further discussions in the subsequent sections. Figure 4.2 provides a dataflow diagram for all components of the SimpliFI implementation. Although all inputs are given to `simplifi.sh`, the diagram depicts when the inputs are actually used.

- **simplifi.sh** – Top level script that controls all functionality.
- **build_tests.sh** – Build all necessary files for a list of tests.
- **build_pgm.sh** – Compile an instruction sequence or application test for the RISC-V ISA.

- **convert_mem.py** – Convert a compiled program into memory initialization data for BRISC-V.
- **build_images.tcl** – Create copies of the device netlist, with a different test program integrated into each one.
- **build_sim.sh** – Build a timing simulation snapshot for a specific test program.
- **gen_config.py** – Generate a fault simulation script from a user configuration file.
- **run_sim.sh** – Simulate all requested fault injection trials for a specific test.
- **sim_ctrl.tcl** – Simulation control script that reads test configurations and configures the testbench.
- **build_lib.sh** – Create a custom copy of the Xilinx simulation libraries that support the metastability model.
- **comp_lib.tcl** – Vivado command to compile the simulation libraries from a custom source.
- **parse_log.py** – Parse the fault simulation log data into a CSV file for future processing with Python3.
- **analyze.py** – Analyze the differences between faulty program data and clean program data to evaluate hardware and software fault responses.

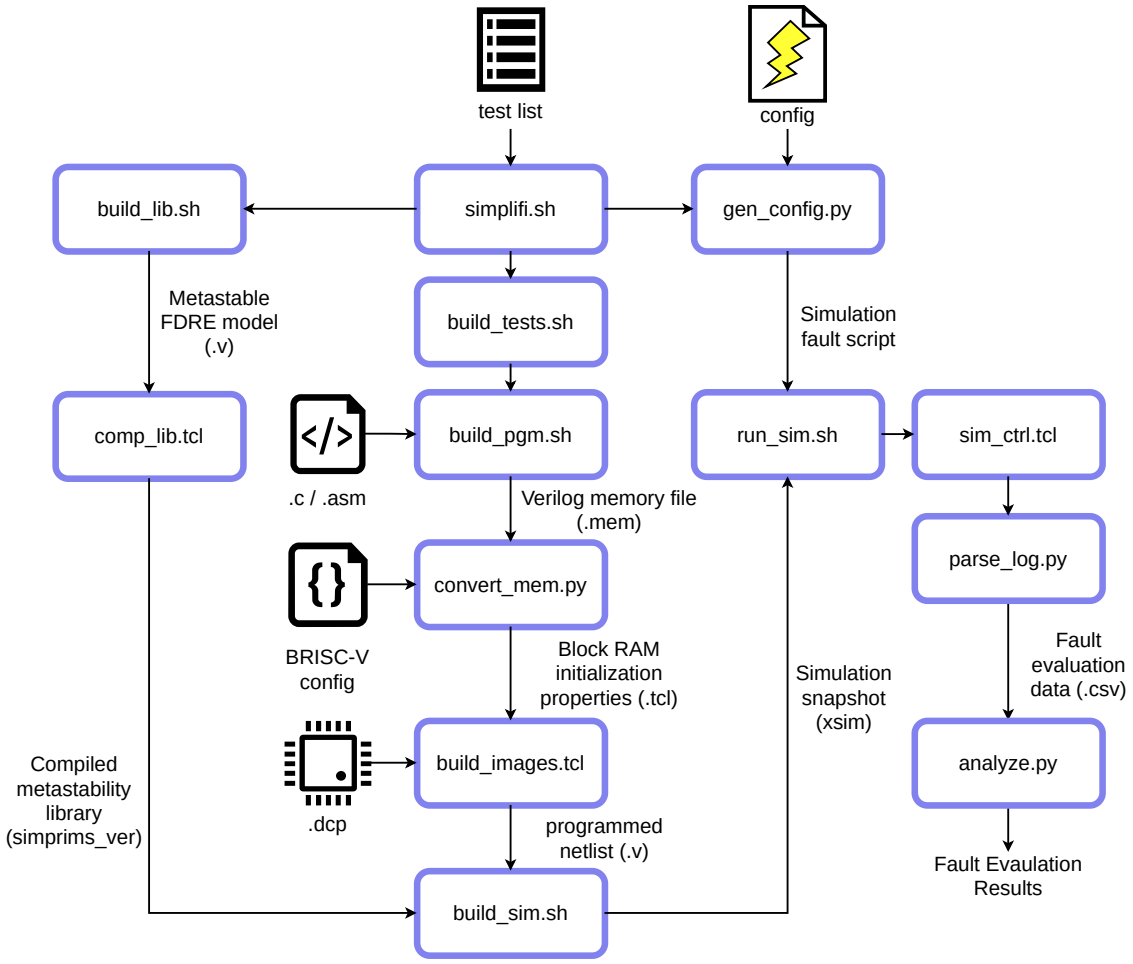


Figure 4.2: Dataflow diagram of the BRISC-V SimpliFI framework implementation.

4.2.1 Top-Level Control

All of the functional steps in the framework implementation are controlled from the top-level `simplifi.sh` script that allows each component to be run individually. This script allows the user to specify the platform they want to use in the case of having multiple devices to test, and specify a file that includes all of the tests that should be evaluated. To improve the organization of tests, the tool supports test folder hierarchies. For example, multiple `ADD` instruction tests that target different instruction components could be located at `tests/add/dst` and `tests/add/src1`. This allows clean test hierarchies. To improve run-time, both the simulation snapshot compilation and simulation execution steps can be parallelized, with multiple tests being built and run at the same time. The practicality of these options is dependent on system resources.

4.2.2 Building Test Cases

To create different test cases the user places programs in a test directory, where the main file for instruction sequences is called `seq.asm`, and the main file for application tests is called `main.c`. The tool will look for an optional accompanying `data.c` file for instruction tests to populate the device memory. This allows the user to initialize the memory for memory instruction tests. For an application, all other `.c` and `.h` files in the test folder are compiled along with the main program. At the end of compilation, the program binary is exported as an ASCII file that contains one instruction encoding per-line, which is used as a Verilog memory specification file for future steps.

To load each test program into the device, the user provides a Xilinx design checkpoint file (`.dcp`) of the post-layout netlist to the tool. For each program, the tool splits the Verilog memory file into separate memory initialization files for all of the Block RAMs present in the BRISC-V processor. BRISC-V splits the entire program and data memory into four memory "modules", where each module provides one byte of a 4-byte instruction word. The number of Block RAMs in each memory byte module is dependent on the memory size. The `convert_mem.py` script does this conversion using the Verilog memory file and a BRISC-V platform configuration file that gives the memory size.

Table 4.1 shows how the memory size affects the number of Block RAMs. The values show the number of 36 Kb Block RAM primitives that are included in each module. The 0.5 values indicate that a single 18 Kb Block RAM is used instead of a full 36 Kb primitive. When a memory module comprises a single Block RAM, each Block RAM contributes one byte to the full memory word. However, in modules comprising two Block RAMs, each Block RAM contributes one nibble (4 bits) to the overall memory word. Finally, in modules comprising one half of a Block RAM, every other byte of the Block RAM contributes to the memory value.

Table 4.1: Mapping of BRISC-V memory sizes to Xilinx Block RAMs.

Memory Size (B)	**Bytes per RAM**	**Bits per RAM**	**RAMs per Module**
32768	8192	65536	2
16384	4096	32768	1
8192	2048	16384	1
4096	1024	8192	0.5

In a netlist that contains initialized Block RAMs, the initial values are specified in initialization parameters of the Block RAM primitive. The tool uses the observed mappings of memory data to Block RAMs to construct appropriate initialization parameters, which are applied to the platform netlist through `set_property` TCL commands. The original Verilog memory file parsing and Block RAM initialization parameter writing is performed by a Python3 script for every test program. A subsequent TCL script is sourced in Vivado to create copies of the device netlist with the various test programs integrated. Each programmed device image is saved both as a new `.dcp` file, and a timing-annotated Verilog netlist.

4.2.3 Simulation Snapshot Compilation

After building a netlist image for each test program, the testbench is compiled into a Xilinx simulation snapshot for each program image. A simulation snapshot is an optimized simulation program that is compiled with all of the design and test files. The tool builds simulation snapshots using a shell script that compiles the netlist Verilog and testbench SystemVerilog files with Xilinx `xvlog`, and elaborates the intermediate result into a simulation snapshot with `xelab`. Since the programmed device netlist is exported with timing information, the `xvlog` commands are not instructed to use further timing information. However, to create to create a timing simulation snapshot with realistic delay times, the following `xelab` arguments are specified:

Listing 4.3: `xelab` command line arguments for accurate timing simulation.

```
1  --relax --maxdelay -timescale 1ns/1ps -O3 -L xil_defaultlib -L
      secureip -L simprims_ver -transport_int_delays --pulse_r 0 --
      pulse_int_r 0 --pulse_e 0 --pulse_int_e 0
```

4.2.4 Simulation Fault Script Generation

As discussed in Section 3.2.1, the user creates a file for each test that specifies the fault parameters and program target locations. For application tests, the user specifies the starting and observation points for fault attacks by calling different macros in the code. The macros shown in Listing 4.4 are designed to create unique execution points in the compiled binary that can be easily identified. For example, when `__SimpliFI_Start` is called in a program, the function will be compiled to the assembly shown in Listing 4.5. The final instruction of this function can only be executed when the function was intentionally entered, and not tentatively entered when executing a branch instruction. The address of the last instruction

is used as the unique identifier for the hardware simulation core, which will begin waiting for the fault target points when this address is seen in the program counter.

Listing 4.4: Macro and corresponding function used to identify the fault injection start point

```
1 void __attribute__((optimize("O0"))) FiSimStart(void)
2 {
3     return;
4 }
5 #define __FiSimStart FiSimStart();asm("nop;");
```

Listing 4.5: Compiled assembly code for the macro in Listing 4.4

```
1 00000264 <SimpliFI_Start>:
2      264: ff010113  addi  sp,sp,-16
3      268: 00812623  sw  s0,12(sp)
4      26c: 01010413  addi  s0,sp,16
5      270: 00000013  addi  zero,zero,0
6      274: 00c12403  lw  s0,12(sp)
7      278: 01010113  addi  sp,sp,16
8      27c: 00008067  jalr  zero,0(ra)
```

As discussed in Section 3.2.1, the user config file is transformed into a fully-specified simulation fault script, where all options for every test are given values. The user configuration file for an application test would include `SimpliFI_Start` as the start point, and the enumerated instructions following the start point that should be tested as the targets. For example, the two instructions immediately following the start point would be selected by adding `0,1` to the target point list. When the tool transforms the user config file into the fault script, it searches for the place that the `SimpliFI_Start` function is called *from*, and uses the instruction two addresses later to expand the target points. If the function is called at address `0000013C`, then the target point corresponding to the `0` in the target list would expand to `00000144`. This is necessary because the testbench needs a buffer cycle in between when the function exits and a fault is injected. This buffer is provided in code by the `NOP` instruction at the end of the function in Listing 4.4.

As a high-level summary, the user specifies the region of the program that should be evaluated by including the macro, and then counts the specific instructions that should be evaluated following the start point. The tool then translates these identifiers into conditions that can be identified with hardware-level simulation.

4.2.5 Testbench and Control

The hardware simulation core testbench discussed in Section 3.2.2 was written in SystemVerilog to allow access to internal device signals, as required by the simulation framework. To support both instruction sequence and full application simulation, the testbench file includes two code bodies which are selected by a `TEST_TYPE` parameter. This parameter is specified during simulation snapshot elaboration. During simulation, the testbench relies on variables that specify the start instruction, target point, and observation points, which are specified by the converted user configuration file. These variables are set from outside the simulation by a TCL control script that reads the simulation fault script. For each test in the script, the TCL control program writes the start and target points into the testbench variables, and repeatedly runs and restarts the simulation with different clock glitch periods. This allows the testbench to fully reset to a clean state for every glitch trial.

While testbenches for system verification usually have a dedicated clock generation process, the SimpliFI simulation clock is manually toggled throughout simulation. While it would be possible to create a clock generation process that is affected by a variable set in the main loop, having one testbench process which handles the clock and program tracking leads to fewer complicated timing events in the testbench.

4.2.6 Metastability Modeling

The metastability model was implemented by modifying the `unisims` library source code to randomize a flip flop's state following a setup/hold violation. The `unisims` flip flop primitives are implemented with timing checks on all of the simulated internal signals to detect when different gate wires have timing violations. The code includes an unused `notifier` variable which raises as a flag when a timing violation occurs. To simulate metastability, the code in Listing 4.6 was added to the `unisims` FDRE (D flip-flop with reset and enable) Verilog file.

Listing 4.6: Code added to the `unisims` library `FDRE.v` file to simulate metastability upon timing violations.

```
1 `ifdef XIL_TIMING
2     reg notifier;
3     wire notifier1;
4     reg rval;
5     always @(notifier) begin
6       rval = $urandom();
7       Q_out <= rval;
8     end
9 `endif
```

In order to use the modified library in Xilinx simulation, the libraries were recompiled with Vivado. This is accomplished from within Vivado using the TCL command shown in Listing 4.7. The `output_path` field specifies the folder where the new compiled libraries are stored, and the `src_path` field specifies where the library source is located. In order for this command to work, the entire Vivado data directory needs to be copied from `Vivado_install_dir/data`, where the location of the copy is given as the source path. Within the library, the modification from Listing 4.6 is applied to the `FDRE.v` file in the `data/verilog/src/unisims` directory . When running simulations with the metastability model included, the `xelab` command line arguments from Listing 4.3 are modified to specify the metastability library by providing the path to `modified_library/simprims_ver`.

Listing 4.7: Vivado TCL command to re-compile the simulation libraries

```
1 compile_simlib -directory <output_path> -family all -language
    verilog -library all -simulator xsim -source_library_path <
    src_path> -no_ip_compile -verbose
```

Chapter 5

Framework Capabilities and Outcomes

This chapter demonstrates SimpliFI's hardware and software fault vulnerability analysis capabilities. The experiments discussed here were performed on the 7-stage pipeline BRISC-V processor using the framework implementation presented in Chapter 4. Both the design and outcomes of the experiments are pertinent. In particular, experiment design is critical for collecting results that maximize insight into hardware fault behavior, which aids the evaluator in understanding the root cause of software-level faults. The chapter first presents the results of instruction sequence experiments, followed by full application results.

5.1 Instruction Sequence Analysis

5.1.1 Experiment Design

Using simple instruction sequences to characterize an embedded processor's fault response has been shown to be an effective technique during physical device testing [22, 23, 24]. SimpliFI supports this technique, and provides benefits over physical testing by enabling rapid evaluation through simulation and collecting hardware state information during execution. The goal of this type of evaluation is to build a knowledge base or model of how any type of instruction may be vulnerable to fault attacks. Therefore, all aspects of an instruction are of interest: opcode, addressing mode, operands, data, etc. An evaluator generally wants to know how each of these instruction components contribute to overall fault response. Acquiring this information essentially provides insight into how the hardware that implements each

component is vulnerable to fault attacks. To fully characterize a device, tests are designed to isolate microarchitectural components to evaluate their fault responses, and to generalize the common fault vulnerabilities and behavior exhibited by instructions with similar parameters.

While this evaluation does not do a characterization of the entire ISA, we divide RISC-V 32-bit integer ISA (RV32I) according to Table 5.1. This organization separates data instructions that use the arithmetic logic unit (ALU), memory access instructions, compare instructions, and branch/jump instructions. Furthermore, each of these categories has instruction variants that use immediate values or direct register values.

Table 5.1: Organization of RV32I by operation type and address/value mode.

Category	**Direct Variants**	**Immediate Variants**
ALU	`SLL, SRL, SRA, ADD, SUB, XOR, XOR OR, AND`	`SLLI, SRLI, SRAI, ADDI, LUI, AUIPC, XORI, ORI, ANDI`
Memory		`LB, LH, LW, LBU,LHU, SB, SH, SW`
Compare	`SLT, SLTU`	`SLTI, SLTIU`
Branch	`JALR`	`BEQ, BNE, BLT, BGE, BLTU, BGEU, JAL`

Within each category, the effect of different destination registers, source registers, source orderings, register values, and immediate values are evaluated. To do this, a separate test is created for each instruction component that isolates the behavior for the component of interest. To test fault effects on the destination register, multiple instructions of the same type are faulted with equivalent values but varying destinations. The same approach is used for the other components of interest. For the purposes of this evaluation, the following terms apply:

- **Instruction Component** – A portion of the specification for an instruction executed by the processor. Examples of instruction components include the destination register, source registers, and immediate values.
- **Instruction Test** – A collection of SimpliFI tests that help characterize the behavior of a specific instruction in response to different fault attacks.
- **Instruction Component Test** – A specific SimpliFI test that is designed to isolate the impact that different microarachitectural blocks have on the instruction fault response by changing one component of the instruction.
- **Fault Response** – The behavior of faulty bits in the processor in response to different fault attacks. The results may refer to the fault response of the hardware state as a

whole, or of the test outputs. For the test outputs, response is usually qualitatively defined in terms of how the number of faulty output bits changes as a function of the fault injection parameter. For example, a **monotonic** output response means that, in general, the number of faulty bits in the outputs consistently increases or decreases. An **oscillatory** output response means that the number of faulty bits alternates between high and low counts as the clock glitch width changes.

- **Fault Sensitivity** – The range of clock glitch widths which induce erroneous bits in the fault response. This term can apply to both the hardware state fault propagation and the test outputs.
- **Fault Intensity** – The number of errors induced in the fault response by a fault injection attack. This term can apply to both the hardware state fault propagation and the test outputs.

The next section gives an in-depth exploration of a subset of the tested instructions to demonstrate how results from SimpliFI can be interpreted. Due to the extensive amount of information collected for each instruction, the rest of the instruction evaluations are summarized briefly. Table 5.2 shows the instructions that are explored in detail. For each instruction component test (eg. destination, source 1, etc.), 4 instances of the same instruction with varying selections of the target component were evaluated. To evaluate each of these components using SimpliFI, a unique test program was written for each instruction component. Listings 5.1 and 5.2 give the specific test programs for two of the `ADD` component tests. The target instructions are padded with `NOPs` to ensure previous and future test setup instructions do not interact with the target instruction execution. Furthermore, each subtest was conducted with faults being applied at each execution stage, with clock glitch widths ranging from 12 to 2 nanoseconds in 250 picosecond increments.

Table 5.2: List of evaluated instruction and their component tests.

Instruction	**Components**
`ADD`	dst, src1, src2, src order, val1, val2, previous dst val
`ADDI`	dst, src, imm., val, val order, previous dst val
`LW`	dst, src, register addr, imm., previous dst val

Listing 5.1: An example instruction sequence test program that focuses onevaluation of the ADD source 1 instruction component.

```
li x23,0x12345678
li x31,0x12345678
li x17,0x12345678
li x2,0x12345678
li x8,0x12345678
(7 nops)
add x5,x23,x8 # Target 1
(7 nops)
add x5,x0,x0  # Clear
(7 nops)
add x5,x31,x8 # Target 2
(7 nops)
add x5,x0,x0
(7 nops)
add x5,x17,x8 # Target 3
(7 nops)
add x5,x0,x0
(7 nops)
add x5,x2,x8  # Target 4
(7 nops)
```

Listing 5.2: An example instruction sequence test program that focuses onevaluation of the ADD value instruction component.

```
li x7,0x12345678
li x8,0xDEADBEEF
(7 nops)
add x5,x7,x8     # Target 1
(7 nops)
li x7,0xFFFFFFFF # Next value
add x5,x0,x0     # Clear
(7 nops)
add x5,x7,x8     # Target 2
(7 nops)
li x7,0xCBDB8576
add x5,x0,x0
(7 nops)
add x5,x7,x8     # Target 3
(7 nops)
li x7,0x23EF89AD
add x5,x0,x0
(7 nops)
add x5,x7,x8     # Target 4
(7 nops)
```

5.1.2 Test Results Exploration

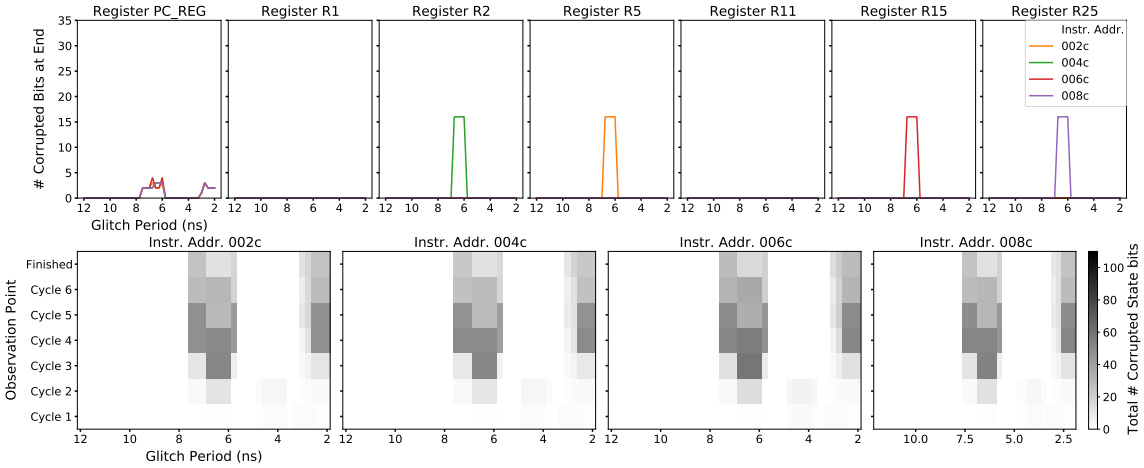
ADD Destination Component Analysis

Figure 5.1 shows the results from injecting clock glitches on four ADD instructions, each with different destination registers. The fault evaluation analysis calculates the number of erroneous bits in register values at the end of execution and in the entire hardware register state throughout execution. The error count is determined by the Hamming distance (HD) between the expected clean results, and the fault injection results. The final instruction outputs and hardware state together offer significant insight into how faults impact instruction execution.

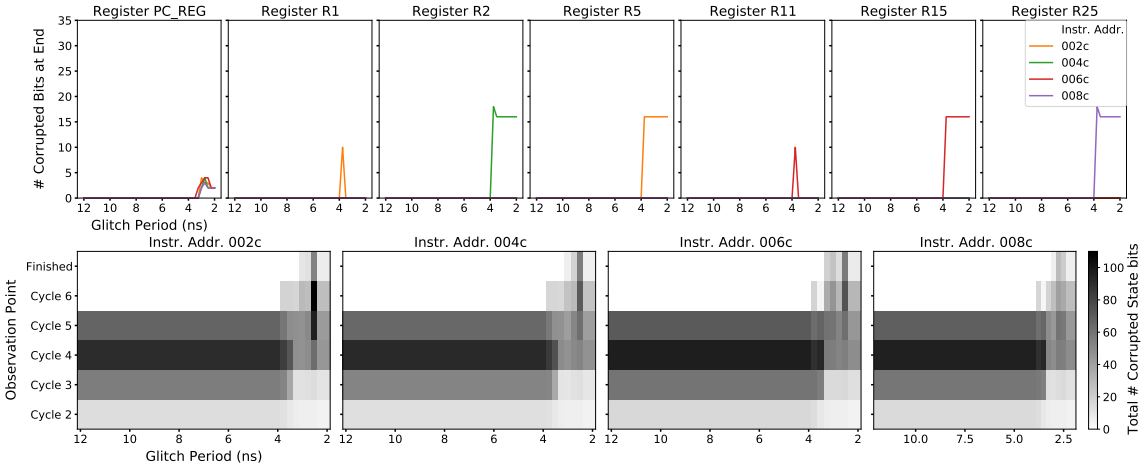
First, the results from each individual stage show that the impact of identical faults on the register outputs vary depending on when the fault is injected. When faults are injected

at the beginning of execution in stage 0, only clock glitch widths between 7.25 and 5.5 ns result in corrupted output data. However, faults injected in stage 2 affect instruction output as long as their glitch widths are shorter than 10.5 ns. In most of the tests, only the targeted destination register and the program counter are affected by fault injection. However, clock glitches with 4.25 ns widths applied in stage 1 not only affect the target destination register, but other registers as well. The instruction targeting register R15 also modifies R11, and the instruction target R5 modifies R1. This behavior is only observed in the stage 1 results in Figure 5.1b.

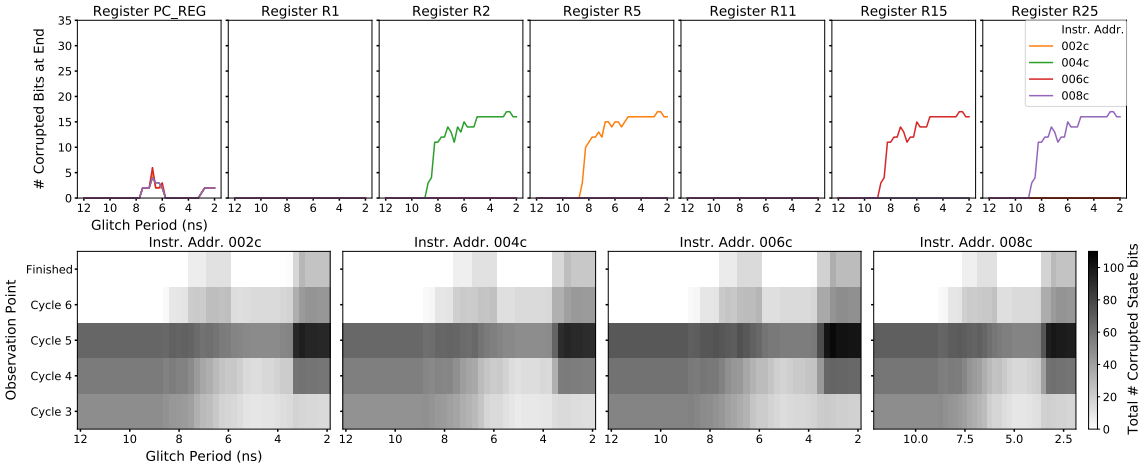
Furthermore, the final erroneous bits in the destination registers vary slightly from one instruction to another. For 4.25 ns glitches injected in stage 1, instructions targeting destinations R2 and R25 result in slightly more erroneous bits. These anomalous behaviors for stage 1 faults call attention to the underlying circuitry for the pipeline stage. It may or may not be a coincidence that targeting registers R5 and R15 results in corrupted values in registers R(5-4=1) and R(15-4=11), while targeting R2 and R25 results in additional corruption in the intended destinations for the exact same fault attack. The important observations here are that *(i)* data can be written to the wrong destination register, and *(ii)* that the selected destination register has a minimal but non-zero impact on the faulty bits that propagate to the end of execution.



(a) Results from faults on stage 0 (instruction fetch).



(b) Results from faults on stage 1 (fetch receive).



(c) Results from faults on stage 2 (instruction decode).

Figure 5.1: Final register corruption and state error propagation for glitch attacks on the `ADD` destination component.

Beyond analyzing the data output behavior, the results from SimpliFI also give us insight into hidden behavior in the hardware. The hardware state fault propagation plots in Figure 5.1 always exhibit erroneous bits remaining in the circuitry up until the end of execution where corrupted output data appears in the registers. For example, a high volume of state corruption can be seen throughout execution in Figure 5.1a in response to faults that cause corrupted outputs. Glitches in the range of 7.75 to 5.75 ns induce hardware state errors that propagate through to the output, although only glitches shorter than 7.25 ns affect the data registers and glitches longer than 7.25 ns only affect the program counter. There is also significant error propagation from glitches shorter than 3 ns, which only affect the program counter by the end of execution.

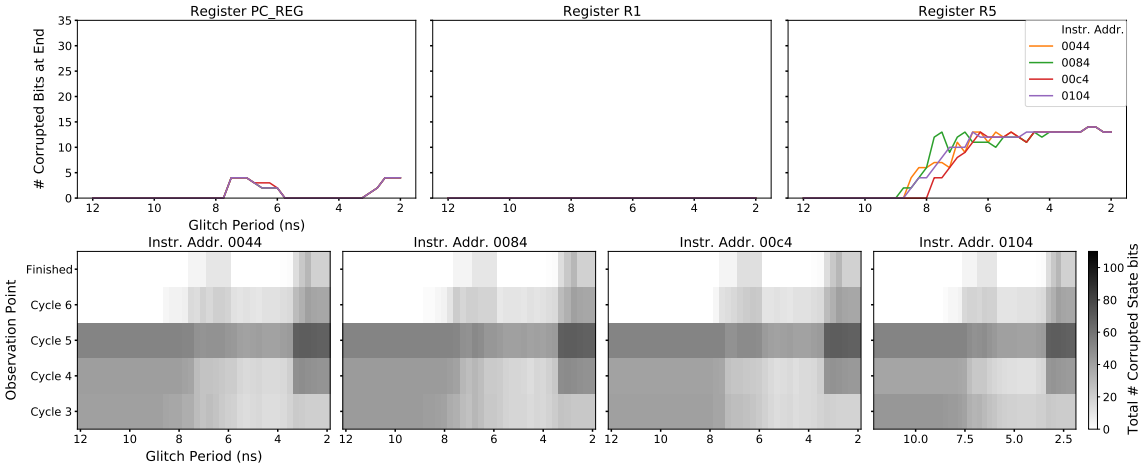
However, attacks during stages 1 and 2 have significantly different effects on the hardware. Figures 5.1b and 5.1c show that every tested glitch width resulted in at least 50 corrupted state bits within the third subsequent execution cycle, but that the glitches applied in stage 1 result in up to 85 state errors. However, none of these errors propagate to the data outputs, and only 7.75 to 5.25 ns glitches in stage 2 resulted in program counter errors.

Another fault behavior in these results is that certain faults induce few errors during injection, while others induce significantly many errors. One trend shown by all of the propagation data is that glitches resulting in few errors at the start tend to cause amplified state corruption a few cycles later. These gradually amplifying faults seem to have the greatest impact on instruction outputs. Conversely, faults that immediately cause significant state error suddenly disappear two clock cycles before the end of execution.

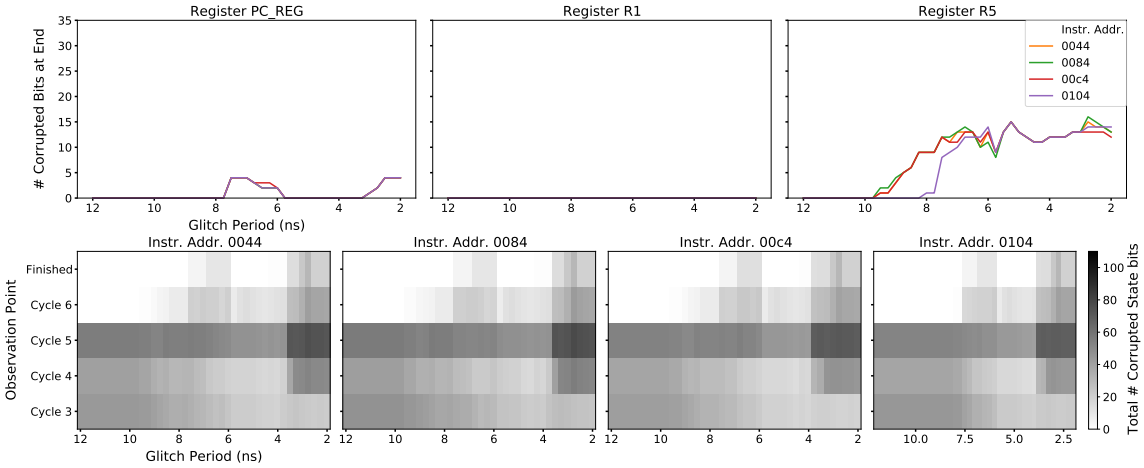
`ADD` Source Components Analysis

Figure 5.2 shows a selection of results from injecting clock glitches on four `ADD` instructions, each with different source registers but identical data. Aside from the data included here, the full results for testing different source registers showed that register R1 is corrupted by 4 ns glitches applied in stage 1. This was similar to the behavior of the stage 1 destination tests. However, the intent of the source operand test is to expose source-dependent fault behavior.

In Figure 5.2, instruction numbers 44, 84, C4, and 104 correspond to testing source operands R23, R31, R17, and R2, respectively. When only varying the first source operand (Figure 5.2a), the final faulty output bits change significantly as the source changes. Although the change in fault response is generally monotonic for all register sources, the patterns exhibited during each source test vary for glitch widths between 9 and 4 ns. On the other hand, the different source register tests exhibit nearly identical corruptions in the program counter, except for a few outliers with source R2.



(a) Source 1 target results from faults on stage 2 (instruction decode).



(b) Source 2 target results from faults on stage 2 (instruction decode).

Figure 5.2: Final register corruption and state error propagation for glitch attacks on the `ADD` source 1 and source 2 components.

The behavior observed when testing source operand 2 is similar, but with some variance, as shown in Figure 5.2a. The faulty outputs from testing instructions with second source operands R23, R31, and R17 have a similar fault response to the source 1 tests, however, the R2 test outputs begin to have errors in response to glitches that are nearly 2 ns shorter than the other source tests. Furthermore, the faulty responses begin to diverge when glitch widths of 3 ns or shorter are applied. The different fault responses observed when testing different source 1 and source 2 operands indicate that the hardware that either pipelines the source selections or that propagates values from the source register to the ALU is unbalanced. The hardware for source 2 appears to be more sensitive since fault responses fluctuate significantly more when targeting the second source operand compared to the first.

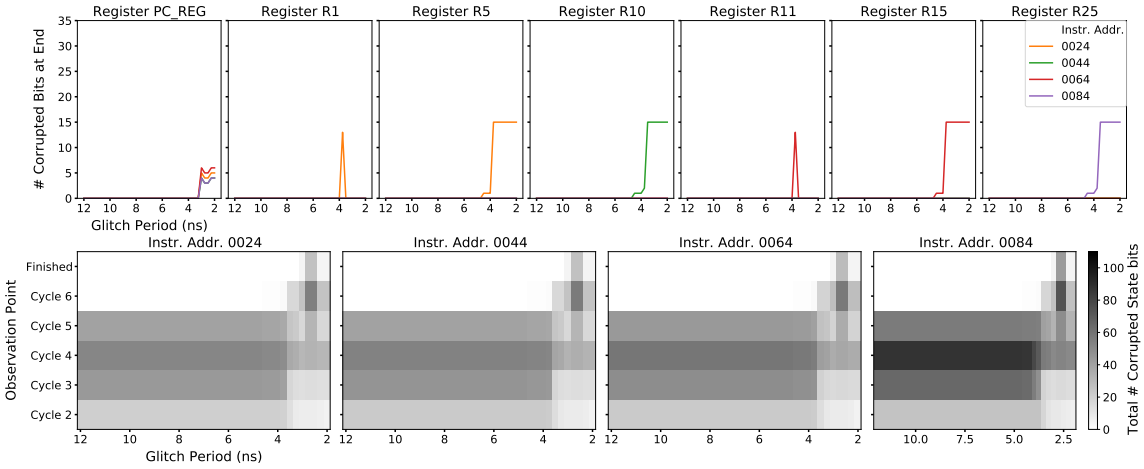
`ADDI` Destination Component Analysis

The `ADDI` instruction tests use the same data and subtest organization as the `ADD` instruction tests, but with the second addend coming from an immediate instruction encoding. The results of the destination component tests in Figure 5.3 show one behavior that is significantly different from the `ADD` destination subtests. Glitches during stages 1 and 2 cause significantly fewer state errors during execution compared to the `ADD` destination tests. However, despite the 20-bit state error decrease in the `ADDI` results, there is no difference in final output corruption.

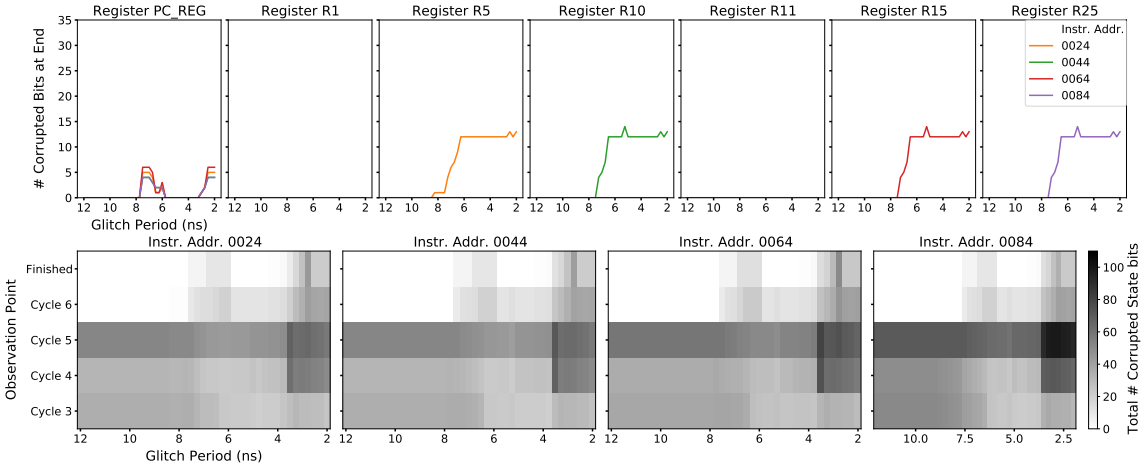
`ADDI` Source and Immediate Component Analysis

The stage 0 injection results in Figure 5.4a for the `ADDI` source register tests show extra hardware fault sensitivities compared to the destination tests. For all previously discussed tests, faults that propagated to the output were induced in the hardware state by glitch widths from 7.5 to 5.75 ns, or 2 to 3.25 ns. However, the left sides of these regions are extended by 0.25 ns in Figure 5.4a for the instructions corresponding to sources R23 and R31, and cause the highest state fault intensity at the end of instruction execution. Despite the significant increase, these faults still only induce errors in the program counter, with no effect on the data registers. Even though this is a marginal increase in the number of faults that actually affect software-level system components, it is significant from a hardware perspective that this extra sensitivity exists for `ADDI` instructions but not `ADD` instructions

The second important result of the `ADDI` source tests is the behavior of faults injected during stage 2. In the `ADD` source tests, both the source 1 and source 2 register fault responses exhibited generally monotonic behavior, with different source 1 registers resulting in varying detailed behavior for glitches less than 4 ns wide. Using different source 2 registers resulted in far less response variance, except for the instruction targeting R2 becoming sensitive to glitches at 8.25 ns instead of 10.25 ns. The stage 2 results of testing different source registers in `ADDI` instructions, shown in Figure 5.4b are similar to the those of the `ADD` instruction source 1 results. On the other hand, the results of using different immediate values in the `ADDI` instructions are somewhat similar to the `ADD` instruction source 2 results. Aside from the `ADD` source 2 response being more chaotic between 5.25 and 6.75 ns, the two component test fault responses share multiple features.



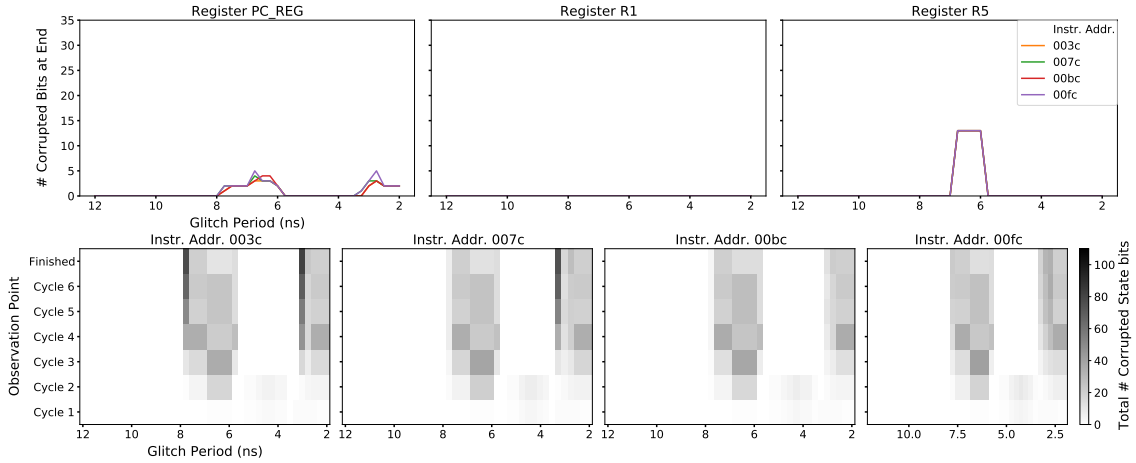
(a) Results from faults on stage 1 (fetch receive).



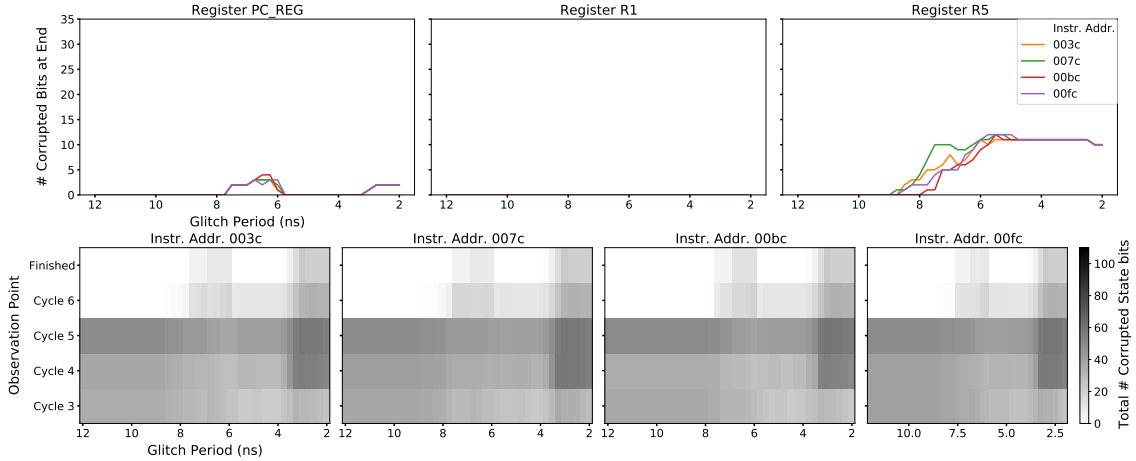
(b) Results from faults on stage 2 (instruction decode).

Figure 5.3: Final register corruption and state error propagation for glitch attacks on the `ADDI` destination component.

The `ADDI` immediate test results in Figure 5.5 show that only glitch widths shorter than 8.25 ns affect the outputs. This is the same behavior exhibited by the `ADD` source 2 results from Figure 5.2b for the instruction targeting R2. Since only one source 2 register resulted in these fault results, no assumption can be made about the similarities to the immediate test results. However, it is likely that the hardware that handles the second operand in ALU operations is responsible for, or related to, the selection between a second source register or an immediate data source. Furthermore, the fault responses for each trial in both of these tests diverge in response to glitches with widths of 3ns or shorter. This behavior was not observed with other instruction components, and is further evidence that the hardware may be shared between direct source 2 registers and immediate values.



(a) Results from faults on stage 0 (instruction fetch).



(b) Results from faults on stage 2 (instruction decode).

Figure 5.4: Final register corruption and state error propagation for glitch attacks on the `ADDI` source component.

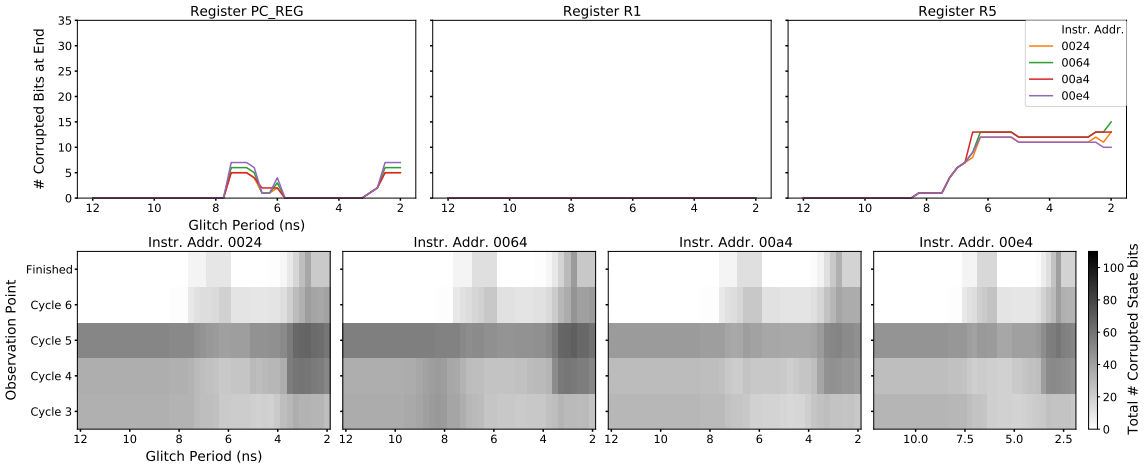
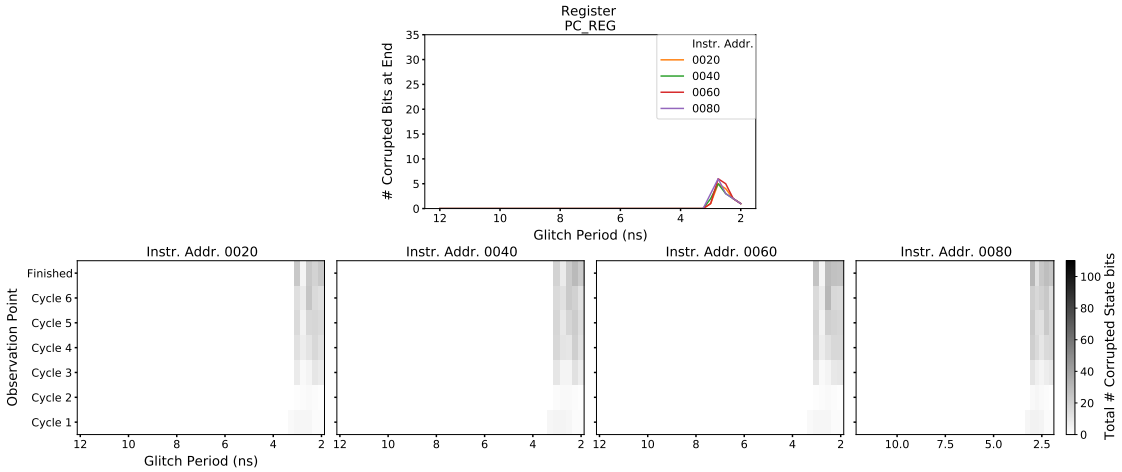


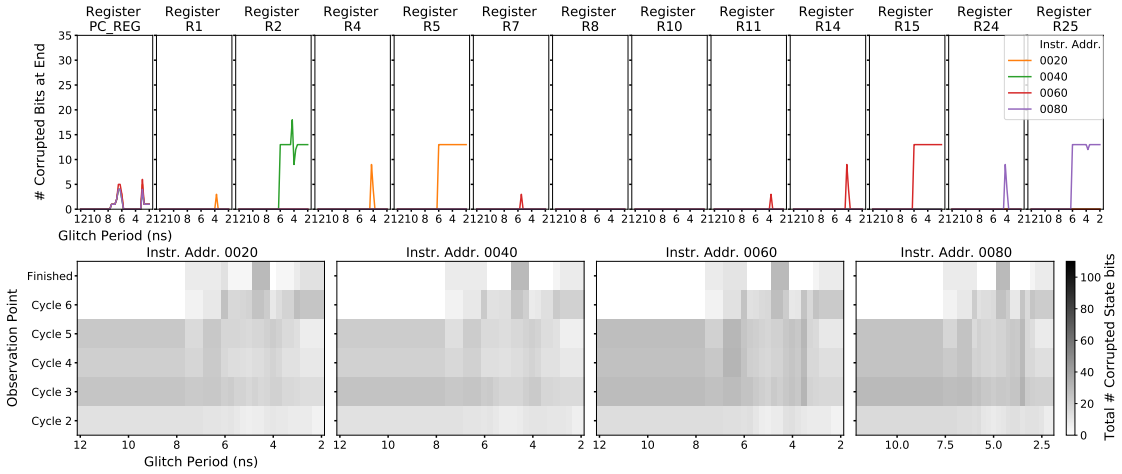
Figure 5.5: Final register corruption and state error propagation for stage 2 (fetch receive) glitch attacks on the `ADDI` immediate component.

`LW` Destination Component Analysis

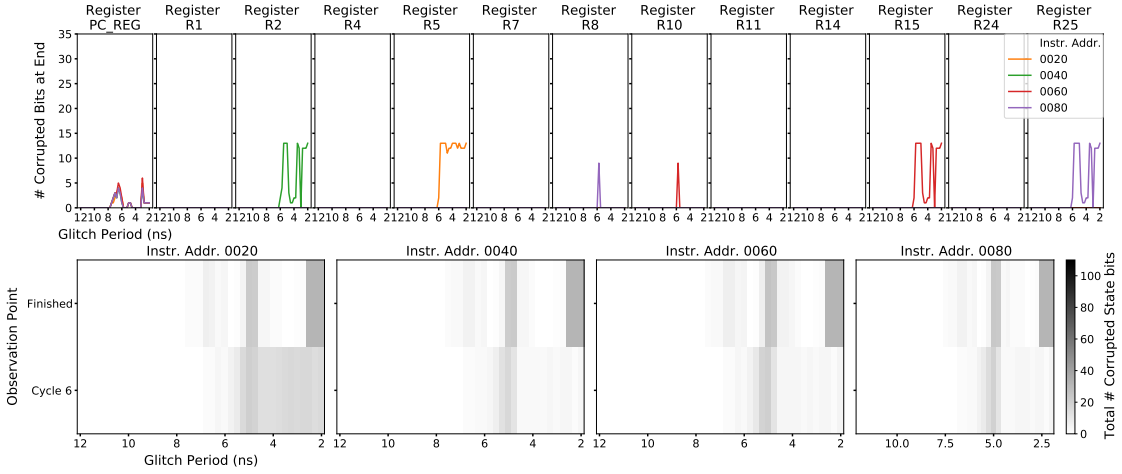
Figure 5.6 depicts the results of fault attacks on the load word (`LW`) destination component. Since no data registers were affected by the clock glitches, they have been left out of Figure 5.6a. The state error propagation due to stage 0 faults only exhibits corruption for glitch widths less then 3.25 ns; compared to the arithmetic instructions' responses, the fault sensitivity region for 6 to 8 ns glitches that causes output errors is missing for the `LW` destination experiment. This difference in stage 0 fault sensitivity was the first significant difference encountered between the two categories, demonstrating that different instruction opcodes influence the hardware fault response. Meanwhile, the stage 1 (Figure 5.6b) error propagation behaves similarly to that of the arithmetic instruction tests. However, each destination test instruction results in a slightly different fault response. In this component test, instructions 20, 40, 60, and 80 targeted destinations R5, R2, R15, and R25, respectively. The responses for R5 and R15 destinations are nearly identical, with both causing errors in two additional registers other than the intended destination. The one exception is the R15 instruction inducing an extra error in register R7. On the other hand, the R2 and R25 tests result in fewer extra corrupted registers and exhibit non-constant bit errors in the expected destination registers.



(a) Results from faults on stage 0 (instruction fetch).



(b) Results from faults on stage 1 (fetch receive).



(c) Results from faults on stage 5 (memory receive).

Figure 5.6: Final register corruption and state error propagation for glitch attacks on the `LW` destination component.

The LW destination test also produced the first faulty outputs induced by a stage 5 glitch, as shown in Figure 5.6c. Furthermore, the destination register fault response exhibits the first oscillatory response seen across all the tests, in contrast to the previous monotonic behavior. However, the R5 test output response does not exhibit the oscillatory behavior, a significant difference in fault response even though the same type of instruction is being tested. While the exact reason is unknown, a possible cause could be registers in the memory subsystem retaining their previous values. Since the R5 test is the first one performed, the memory output transitions from 0 to the stored value, while in subsequent target instructions, the memory output may already be populated with the correct value. Furthermore, the R5 output errors still ripple in the same spots where the other test responses drop close to 0, so the same paths are likely being violated.

When testing different values being loaded from the memory using the same address, source, and destination, all instruction responses exhibit the oscillatory behavior (Figure 5.7). One significant difference between the instruction sequences used for the value and destination tests, is that in the value test, a new value is stored in between the target LW instructions, but in the destination test no stores are performed. With this information, another guess that can be made about the memory subsystem is that store instructions clear the state that was responsible for the first instruction having different behavior in the destination tests. Once again, targeting multiple components of the instruction uncovers unexpected behavior in the device microarchitecture.

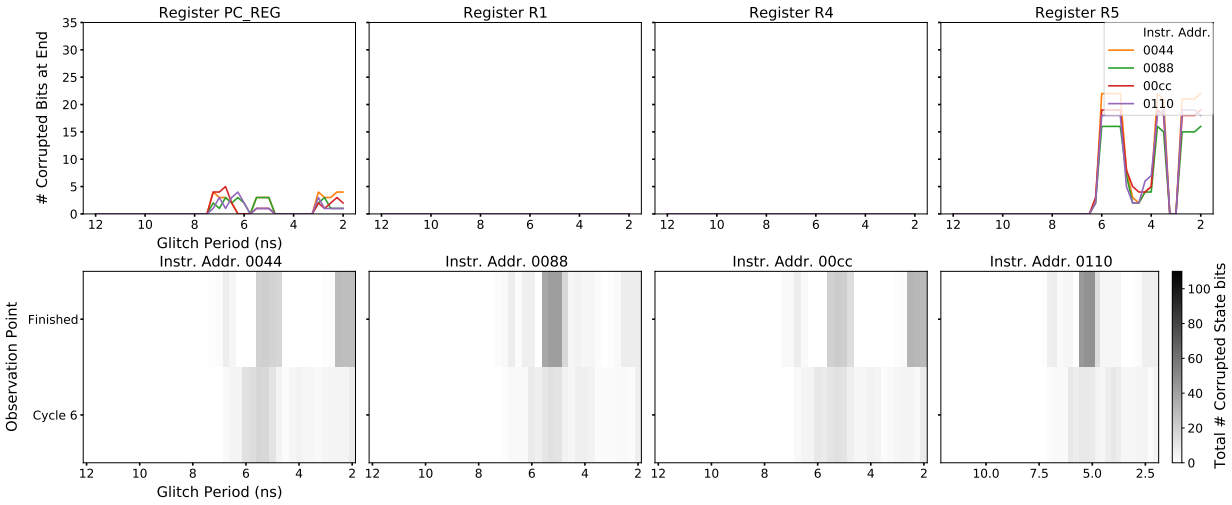


Figure 5.7: Final register corruption and state error propagation for stage 5 (memory receive) glitch attacks on the LW memory value.

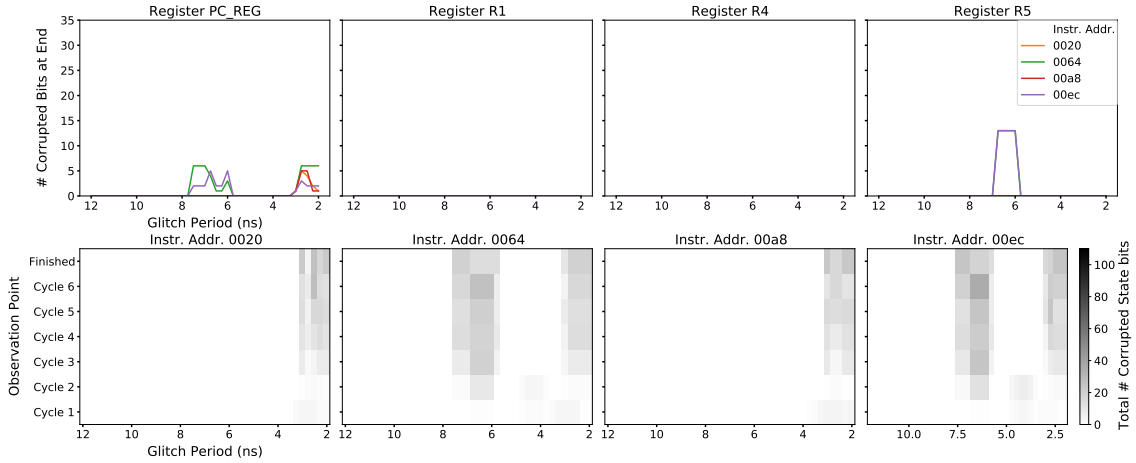
`LW` Address and Immediate Component Analysis

The last set of tests target the address and immediate components of the `LW` instruction. The memory address is read from a source register and has an offset added to it which is specified by the immediate field. The address tests accessed different memory locations by changing the value in the source register and keeping the immediate value at 0. When testing the immediate field, the value in the source register was fixed to point to the start of RAM and the immediate field was changed for each target instruction to access different addresses. All of the memory addresses accessed in these component tests held the same values.

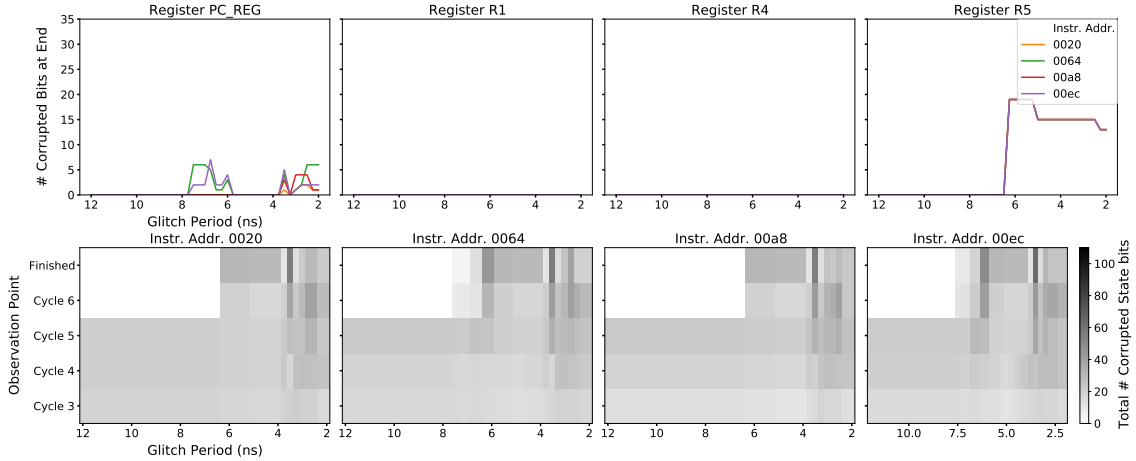
The first notable behavior is that faults during stage 0 always result in the same hardware fault propagation when varying the immediate field (Figure 5.9a), but cause different hardware propagation depending on the address (Figure 5.8a). The hardware fault intensity of the immediate tests matches that of the destination component tests in stage 0, which is not surprising since the same type of load instruction is being used. However, it is unknown why using different address values in the same source register would have an impact on fault manifestation during the instruction fetch, since the source register value is not in the pipeline during the fetch. When using SimpliFI to evaluate a processor for production, anomalies like these may indicate unintended microarchitectural impacts that should be addressed.

The outcomes of stage 2 faults for the immediate tests in Figure 5.9b show that the hardware fault sensitivity regions are dependent on the immediate value being used. For instructions 20 and 60, the sensitivity region for faults that propagate to the output extends out to 6.25 ns, while the other instruction outputs are only affected starting with 3.75 ns glitches. In contrast, changing the address for memory access has no impact on the output fault response, as shown in Figure 5.8b. Although the hardware error intensity varies, the output sensitivity does not change significantly. The only change in fault response caused by these faults appears in the program counter, whose error variance is partially caused by the instruction address increasing as later `LW` instructions are tested. The difference in behavior for targeting the address versus the immediate value indicates that, for `LW` instructions, the instruction decode stage circuitry dedicated to the immediate field is more sensitive to glitches than the circuitry that holds the source register value.

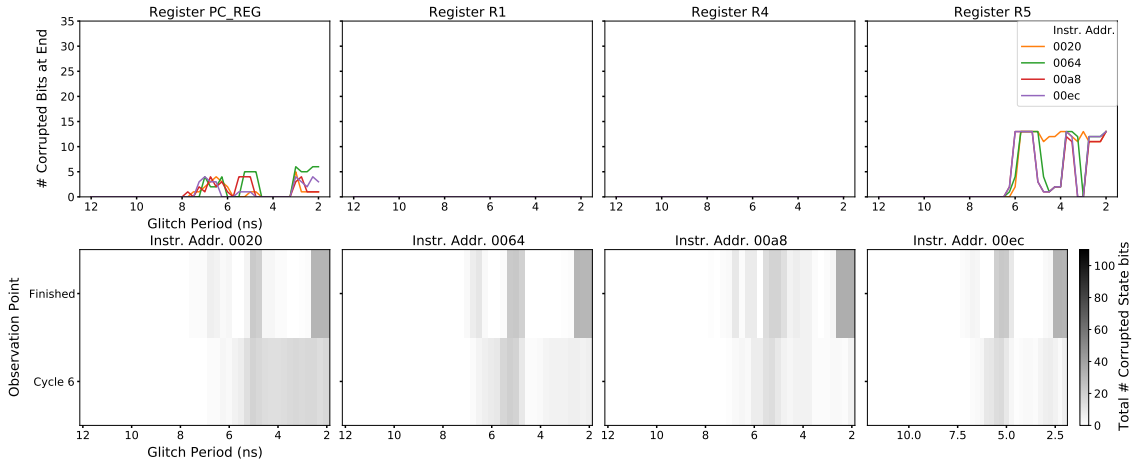
Finally, both the address and immediate results exhibit the same fault responses for stage 5 faults, shown respectively in Figures 5.8c and 5.9c. At this point in the execution, the memory access address has been fully computed using the source register and the immediate field. Whether the address was mostly determined by the source or immediate has no effect on the execution beyond this point. This aspect of the processor is demonstrated by the fault propagation plots having identical behavior for both the address and immediate component tests. The only variance seen in these trials is in the program counter, whose behavior is dependent on the instruction address. Since the target instructions for both component tests are stored at different addresses, the program counter variance is expected.



(a) Results from faults on stage 0 (instruction fetch).

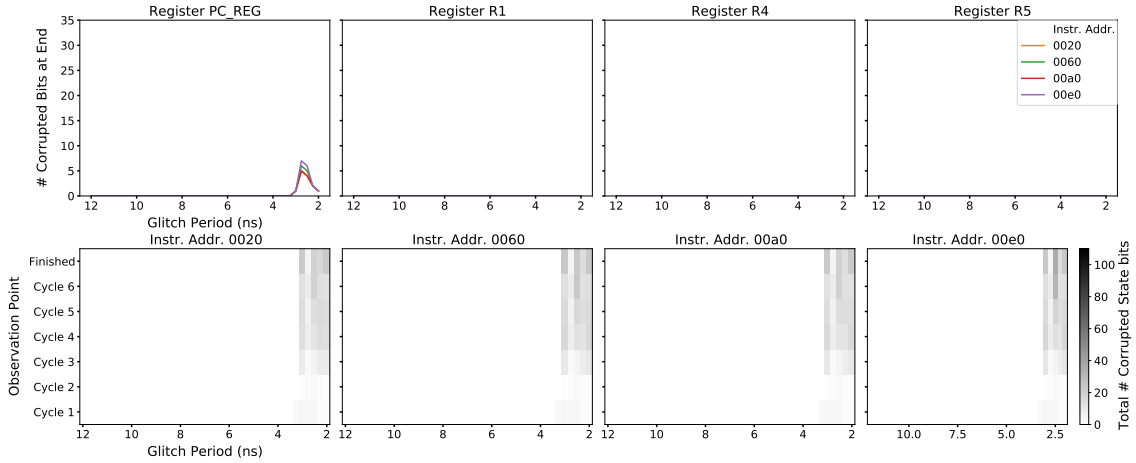


(b) Results from faults on stage 2 (instruction decode).

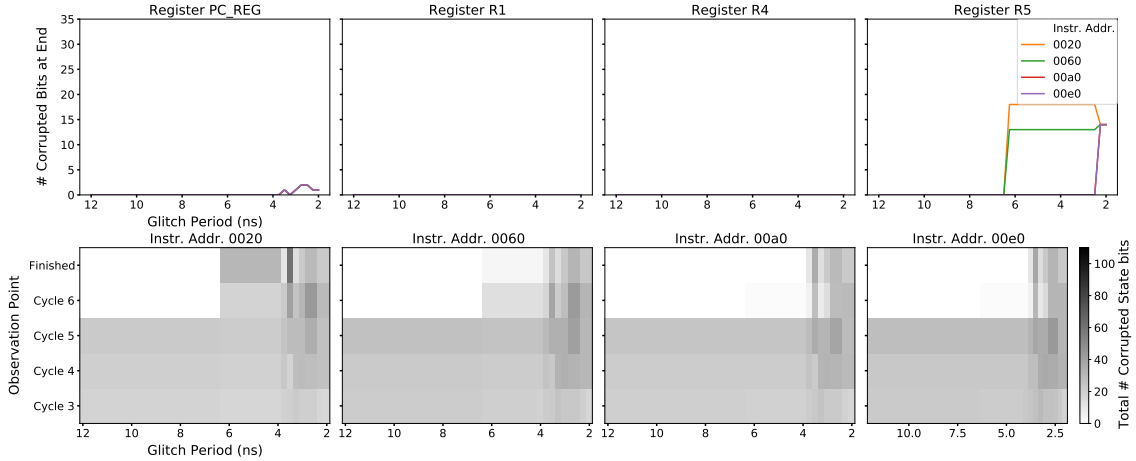


(c) Results from faults on stage 5 (memory receive).

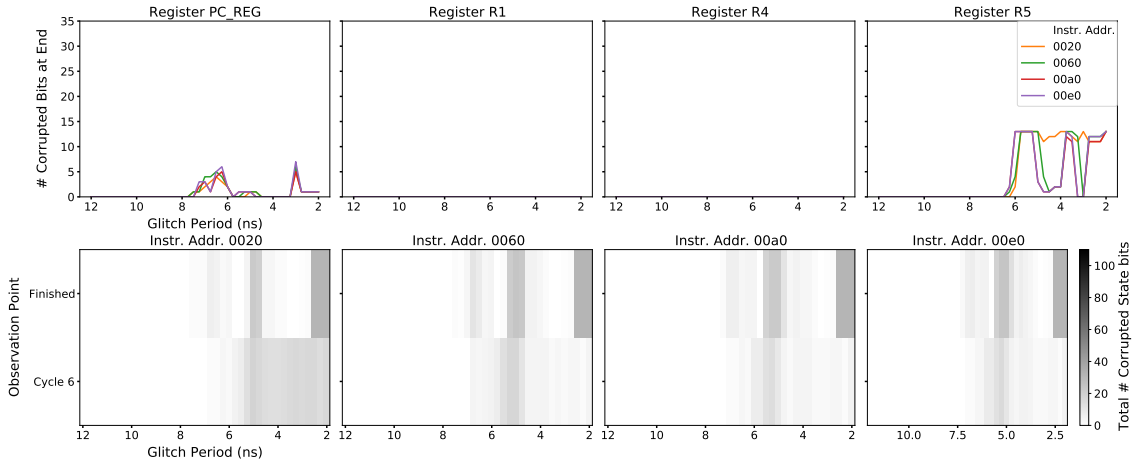
Figure 5.8: Final register corruption and state error propagation for glitch attacks on the `LW` address component.



(a) Results from faults on stage 0 (instruction fetch).



(b) Results from faults on stage 2 (instruction decode).



(c) Results from faults on stage 5 (memory receive).

Figure 5.9: Final register corruption and state error propagation for glitch attacks on the `LW` immediate component.

5.1.3 Test Results Summary

Testing individual instructions with SimpliFI can reveal many microarchitectural intricacies that contribute to software-level fault responses. Based on processor design principles, different classes of instructions are expected to have different fault responses due to using different subsets of the hardware or using the same hardware in different ways. While the final register values give sufficient information for exploring fault effects on instruction results, the information collected on hardware fault propagation reveals further processor intricacies that may be unexpected. The following is an overview of the software and hardware behavior of fault attacks on the `ADD`, `ADDI`, and `LW` instructions.

ADD Instruction

- The destination register has a minimal effect on the output fault response.
- Faults in stage 1 can corrupt unexpected registers when certain destinations are used.
- Faults in stage 1 cause significantly more errors during execution compared to other stages, but few faults affect the output.
- The source 1 register has a significant impact on faulty outputs for [9,4] ns glitches in stage 2.
- The source 2 register has minimal impact on faulty outputs for [9.75,3] ns glitches in stage 2.
  - Source R2 exhibits a different response than other registers for [9.75,6.5] ns glitches.
- The source 2 register has significant impact on faulty outputs for [3,2] ns glitches in stage 2.
- The number of errors in the output register changes monotonically with the change in glitch width.

ADDI Instruction

- The destination register has a minimal effect on the output fault response.
- Faults in stage 1 can corrupt unexpected registers when certain destinations are used.
- Some destination registers cause higher hardware fault intensity for stage 1 faults, although they do not propagate to the output.
- The source register has no impact on the output fault response for stage 2 faults.

- Some source registers cause increased hardware fault propagation for 7.76 and 3.5 ns glitches in stage 1, but no change in the output response.
- The immediate value has minimal impact on the output fault response for [8.25,3] ns glitches in stage 2.
- The number of errors in the output register changes monotonically with the change in glitch width.

LW Instruction

- The first instruction tested exhibits a different fault response potentially due to the memory subsystem design.
- If a value is written to the memory before the first load instruction, the fault response of the first instruction more closely resembles the responses of the other instructions
- Faults in stage 1 can corrupt unexpected registers when certain destinations are used
  - Targeting R15 can result in up to 3 unrelated registers being corrupted.
- The value in the memory address source register appears to affect the fault response for stage 0 faults.
- The immediate field has no effect on the fault response for stage 0 faults.
- The immediate field and address components have different effects on the fault response for stage 2 faults.
  - The immediate value causes different glitch width sensitivities in the hardware.
- By stage 5, the immediate field and address affect the fault response equally because they are combined into a single lookup address.
- The number of errors in the output register oscillates as the glitch width changes.

Cross-Category Observations

- Changes in the immediate value for `ADDI` have similar impacts on fault response as the source 2 register in `ADD`.
- The program counter is generally affected by [8,6] and [3.25,2] ns glitches, but not other glitch widths.
- The output fault responses to stage 6 faults are similar for all tested instructions.
- Attacks on `LW` instructions cause fewer state errors than `ADD` and `ADDI` instructions.

5.2 Full Application Analysis

5.2.1 Experiment Design

While analyzing the fault response of individual instructions is an effective method for evaluating microarchitectural fault vulnerabilities, the goal for software-level analysis is to determine the program outcomes from realistic faults. Evaluating application-level fault responses can be done directly with SimpliFI by simulating full program execution, as opposed to just a few instructions. Instead of observing the behavior of individual registers across multiple fault injections, the final program output values and runtime are measured and compared to the expected results from a clean execution. The faulty program behavior is placed into one of six categories:

1. **Silent Fault** – The hardware state was not affected by the fault.
2. **Unsuccessful Fault** – The hardware state was affected, but caused no change in program behavior.
3. **Fatal Error** – The program did not complete within 500 clock cycles beyond the expected execution time.
4. **Output Corruption** – The program produced faulty outputs.
5. **Time Difference** – The program execution time was different than expected, but no outputs were affected.
6. **Output and Time Corruption** – The program both produced faulty outputs and executed in an unexpected number of cycles.

The results presented in this section are from the evaluation of an Advanced Encryption Standard (AES) program running on the BRISC-V processor. Differential Fault Analysis (DFA) attacks on AES have been shown to be effective for a number of points in the algorithm. Two possible points are after the first add round key operation, and during the last round of encryption before the mix columns operation [30, 31]. The implementation tested is an unprotected, t-table-based version of the MbedTLS library. The implementation was obtained from the NIST lightweight cryptography benchmarking suite, which uses this AES implementation as a baseline for evaluating cipher performance [32].

As discussed in Chapter 4, the code was instrumented for SimpliFI fault simulation by calling the `__SimpliFI_Start` and `__SimpliFI_Observe` macros to mark the start of the injection point region and final output observation point, respectively. Two points in the

algorithm were targeted: the input of the first round following the round 0 add key transformation, and at the start of the 9th round. The instructions around these points heavily consist of arithmetic and logic instructions and only 1 or 2 memory instructions, as shown in Listings 5.3 and 5.4. Both points in the program have a control flow instruction, where round 1 has a branch-on-greater-than instruction and round 9 has a jump-and-link instruction. Clock glitches were injected at the start of each instruction, with glitch widths ranging from 12 to 2 ns.

Listing 5.3: Targeted instructions for AES round 1 fault attacks.

```
ee4: lw    s11,0(s2)
ee8: srai  s11,s11,0x1
eec: addi  a5,s11,-1
ef0: sw    a5,12(sp)
ef4: bge   zero,a5,16cc
ef8: addi  s5,a5,0
efc: lui   a5,0x1
f00: addi  a6,s7,0
f04: addi  a5,a5,1992
f08: andi  s8,s10,255
f0c: srli  t4,s4,0x6
f10: srli  t3,s4,0x18
```

Listing 5.4: Targeted instructions for AES round 9 fault attacks.

```
12e4: jal   zero,f04
12e8: lw    a5,12(sp)
12ec: slli  a5,a5,0x5
12f0: add   s7,s7,a5
12f4: srli  t2,s9,0x18
12f8: srli  a1,s10,0x18
12fc: slli  a5,t2,0x2
1300: slli  a1,a1,0x2
1304: add   t2,s1,a5
1308: srli  a2,s10,0x6
130c: add   a1,s1,a1
1310: srli  a4,s11,0x18
```

5.2.2 Test Results Exploration

A simple breakdown of the test outcomes for attacks on rounds 1 and 9 of the AES implementation is given in Table 5.3. In general, round 1 attacks caused more fatal errors than round 9 attacks, while round 9 attacks caused more output corruption with and without time differences. This is intuitive from an algorithmic point of view, since faults injected earlier in the program have a longer amount of time to cause more errors throughout execution. The plots in Figure 5.10 show much more detail about which exact faults caused these outcomes. Comparing the vertical patterns to the horizontal ones reveals that different glitch widths tend to have similar effects on program execution no matter which instruction they were applied to. Conversely, the impact of different faults applied to the same instruction varies significantly as the glitch width changes.

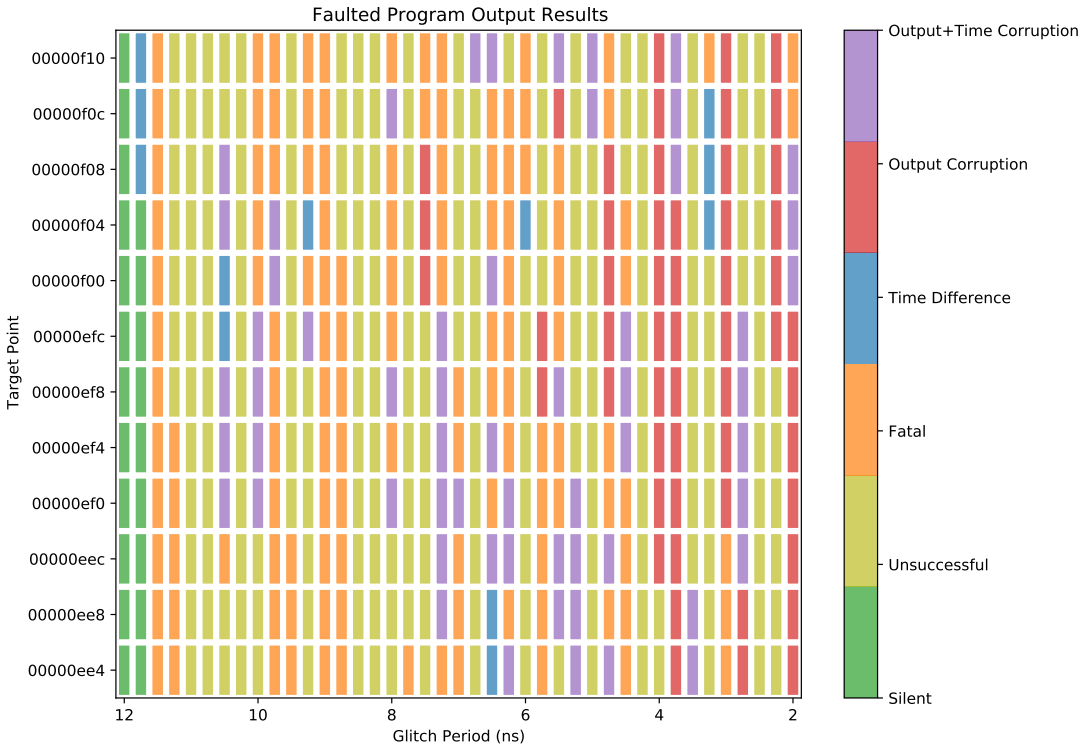
The results in Figure 5.10 also show how the same glitches have different effects on rounds 1 and 9. While the instructions executed during each point are slightly different, there is

Table 5.3: Breakdown of AES fault simulation results by outcome category.

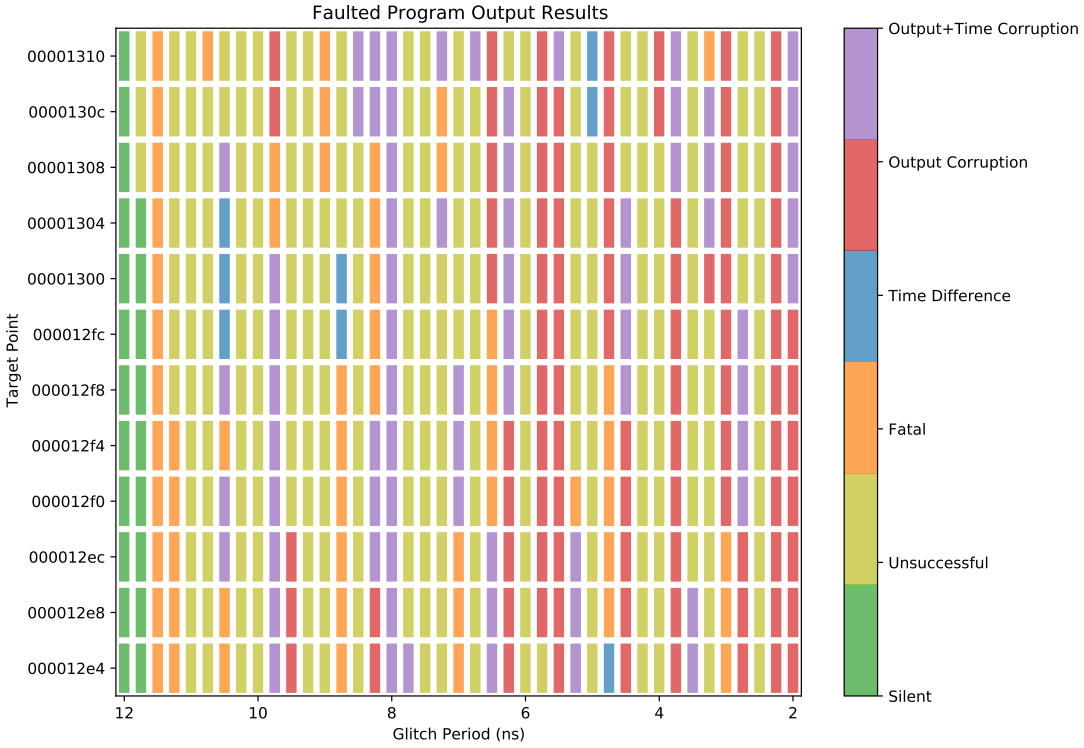
Location	% of Total Test Outcomes					
	Silent	**Unsuccessful**	**Fatal**	**Time Difference**	**Output Corruption**	**Output+Time Corruption**
Round 1	4.27	45.12	25.40	2.44	10.98	11.79
Round 9	4.27	49.19	11.38	1.63	18.90	14.63

enough similarity to compare the results. For example, the [12,10] ns range has nearly the same effect when applied both rounds, except for a few extra execution time inconsistencies and fatal errors at 11.75 and 10.5 ns. Some interesting features of how the results differ are seen at [9.75,9.5] ns and [8.5,8] ns. A two-column region of fatal errors at 9.75 and 9.5 ns in Figure 5.10a turns into mostly output+time corruption outcomes in Figure 5.10b. However, next to this at 9.25 and 9.0 ns, another region of fatal errors turns into mostly unsuccessful faults. Similar to the first region, the unsuccessful faults from 8.5 to 8.0 ns in the round 1 tests turns into mostly output+time corruptions with a few fatal errors.

The behavior when faulting round 9 is ideal for an attacker compared to round 1. Since the adversary wants to collect faulty outputs, the round 1 fatal errors are undesirable. In round 9 the attacker would have a greater chance of obtaining faulty outputs and less of a chance of crashing the device. This insight into the program behavior is valuable for software engineers since it can inform them about which points in the program are more vulnerable. With these results, the software engineer would likely focus more time securing the round 9 operations, and focus less on round 1 due to the high number of fatal errors. Of course, this is dependent on the application; it may still be important for a particular system to prevent fatal errors from occurring even in round 1.



(a) Results from faults on round 1.



(b) Results from faults on round 9.

Figure 5.10: Program impacts of glitch attacks on the starting instructions of different AES rounds.

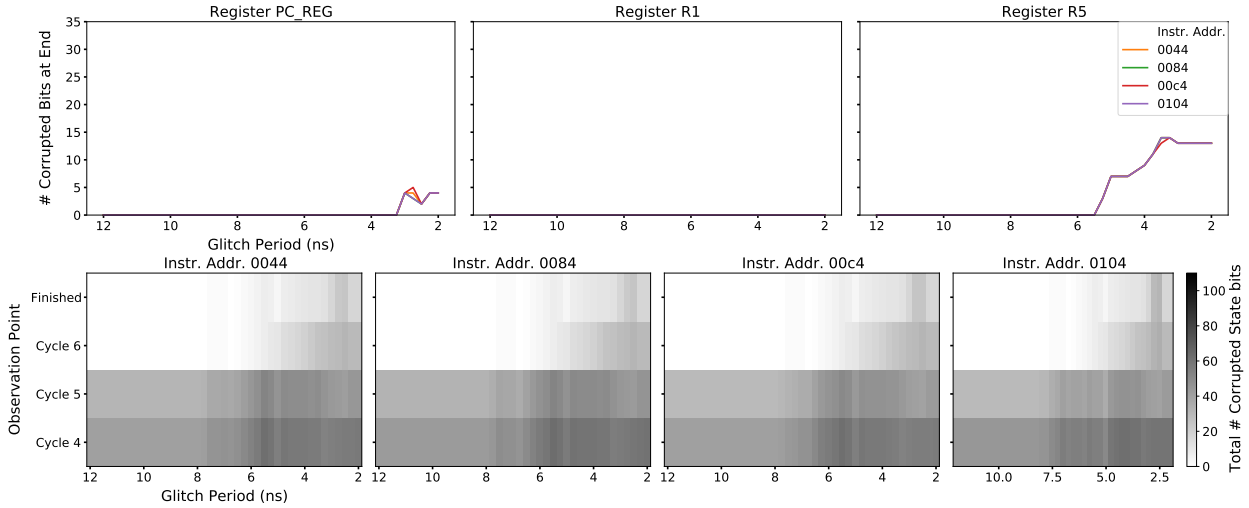
5.3 Effects of the Metastability Model

Each of the experiments explored in this chapter were also conducted using the random bit assignment metastability model implemented in SimpliFI. The results collected with the model in use do not vary significantly for either the instruction or application tests, however they demonstrate that low-level physical effects are not easy to capture in simulation. The clock glitch fault mechanism leveraged in SimpliFI does simulate realistic timing-violation-based fault manifestation, however there are still underlying physical effects that are difficult to capture, such as clock skew and noise. The metastability model is one way to include the effects of physical phenomena that cannot be achieved in gate-level simulation with high accuracy. The goal of SimpliFI is to integrate hardware and software fault evaluation using the most realistic faults that can be reasonably simulated while still having access to software-level activity. The metastability results emphasize that physical phenomena can be taken into consideration with digital simulation if there is some sort of mechanism to emulate it. The metastability model demonstrates that fault results do depend on unpredictable physical events, and acts as a first step towards building even more-accurate simulations.

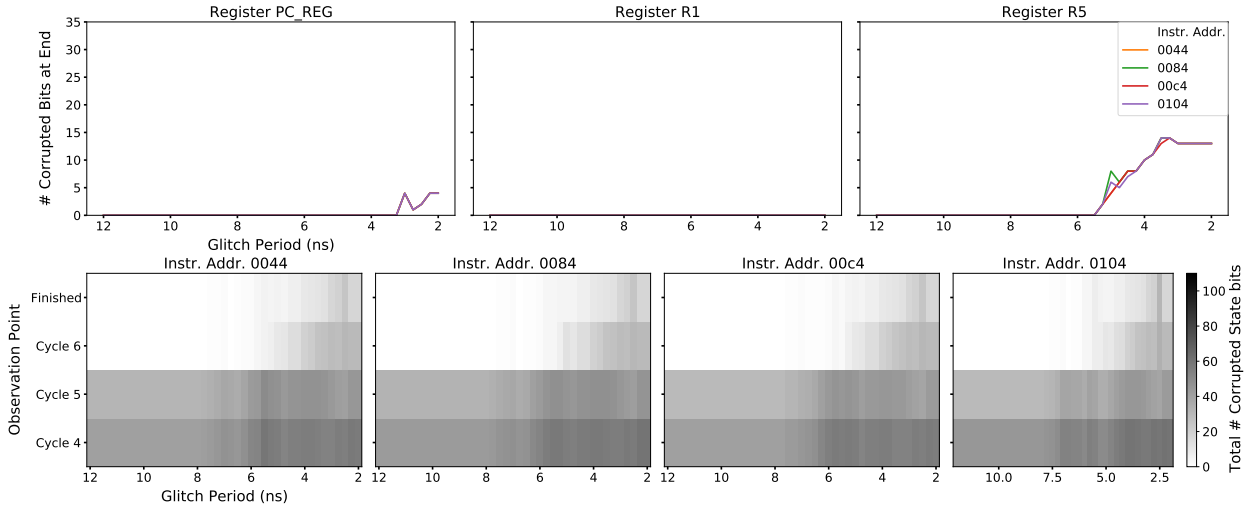
Instruction Test Effects

The metastability model had varying effects on the instruction sequence tests, with arithmetic instructions having minimal differences in outcomes, and memory instructions having significant differences. The `ADD` instruction tests on the source 2 component during stage 3, shown in Figure 5.11, are a good representation of the behavior of the `ADD` and `ADDI` instructions tested. In the standard tests, the number of output errors is nearly identical for all the second source registers tested. However, the output errors for each second source register with the metastability model active exhibit more linear segments and sharper peaks.

The results for stage 1 of the `LW` destination tests exhibit much more extreme differences from applying the metastability model. First, the sudden spikes in the instruction 40 (green) destination register in Figure 5.12 have greater amplitude with the metastability model. While this is the main difference for that particular instruction which used R2 as the destination register, instructions 20, 40, and 80, respectively targeting registers R5, R15, and R25 were impacted more significantly. Without the metastability model, faults on instruction 60 only corrupted 3 extra data registers, while the metastability tests corrupted 4 extra data registers. The original instruction 60 trials did not corrupt R8 in this stage, while the metastability trials induced nearly 30 bit errors in the final R8 value. The error counts in some registers during instructions 20 and 80 were around this as well. These drastic results from the `LW` test further stress the importance of taking hardware-level fault manifestation into consideration for software fault evaluation. Adding the metastability model revealed additional fault responses caused by feasible random phenomena that were not discovered during the original tests.

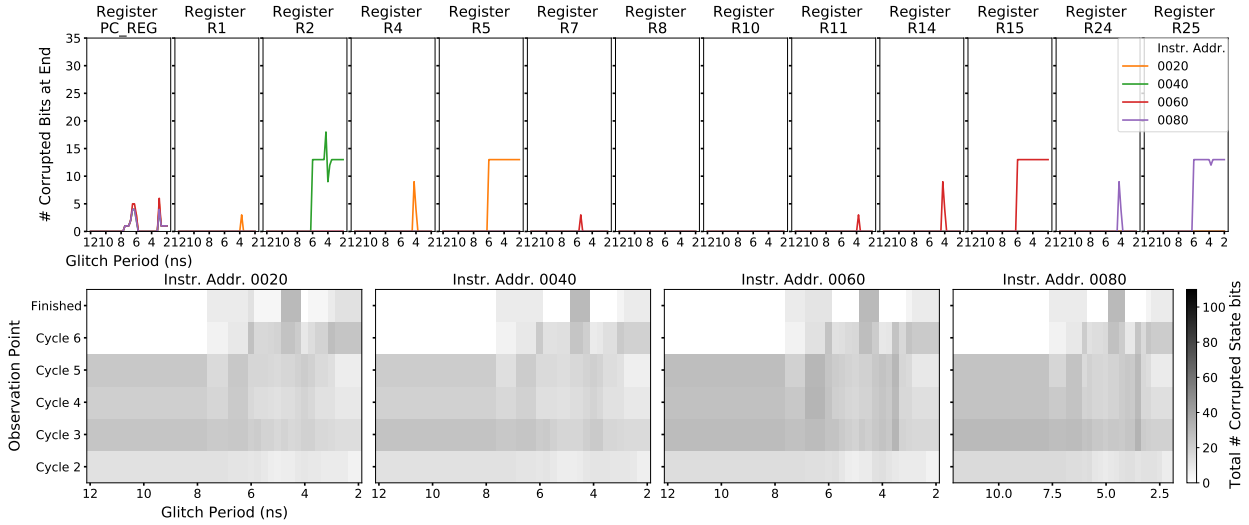


(a) Results without metastability model.

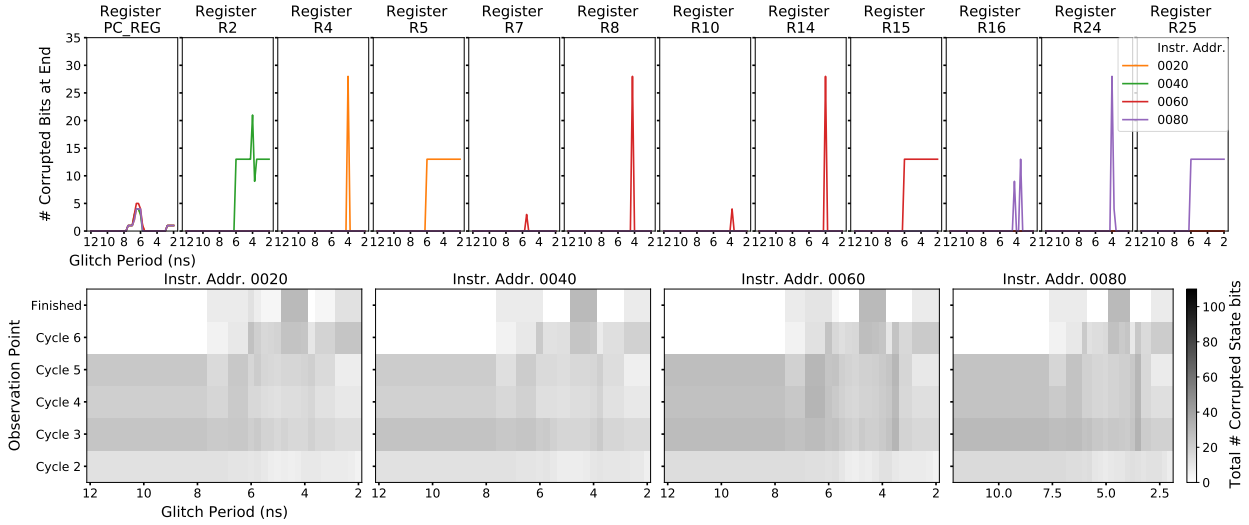


(b) Results with metastability model.

Figure 5.11: Final register corruption and state error propagation with and without the metastability model for stage 3 (execute) glitch attacks on the `ADD` second source register.



(a) Results without metastability model.



(b) Results with metastability model.

Figure 5.12: Final register corruption and state error propagation with and without the metastability model for stage 1 (fetch receive) glitch attacks on the `LW` destination register.

Application Test Effects

Table 5.4 shows the change make-up of the faulty AES program outcomes. These percentages were calculated by taking the difference between percentages of the total outcomes for the standard experiments and the experiments using the metastability model. The locations of the round 1 metastability test outcomes that differ from the original results are shown in Figure 5.13. Surprisingly, applying the metastability model to the round 9 faults did not

affect any of the program outputs. Fault outcomes that did not change from the original results are blanked out, leaving on the outcomes that changed between tests. According to the table, the occurrences of output+time corruption outcomes when targeting round 1 decreased more than any other outcomes increased. Since new output+time corruption outcomes can be seen in the Figure 5.13, many of the other changed outcomes must have originally been output+time corruptions in order to achieve the -1.22% decrease.

Table 5.4: Change in Breakdown of AES fault simulation results after adding the simulated metastability model.

	Change in % of Total Test Outcomes					
Location	**Silent**	**Unsuccessful**	**Fatal**	**Time Difference**	**Output Corruption**	**Output+Time Corruption**
Round 1	0	0.21	0.82	0.20	0.00	-1.22
Round 9	0	0	0	0	0	0

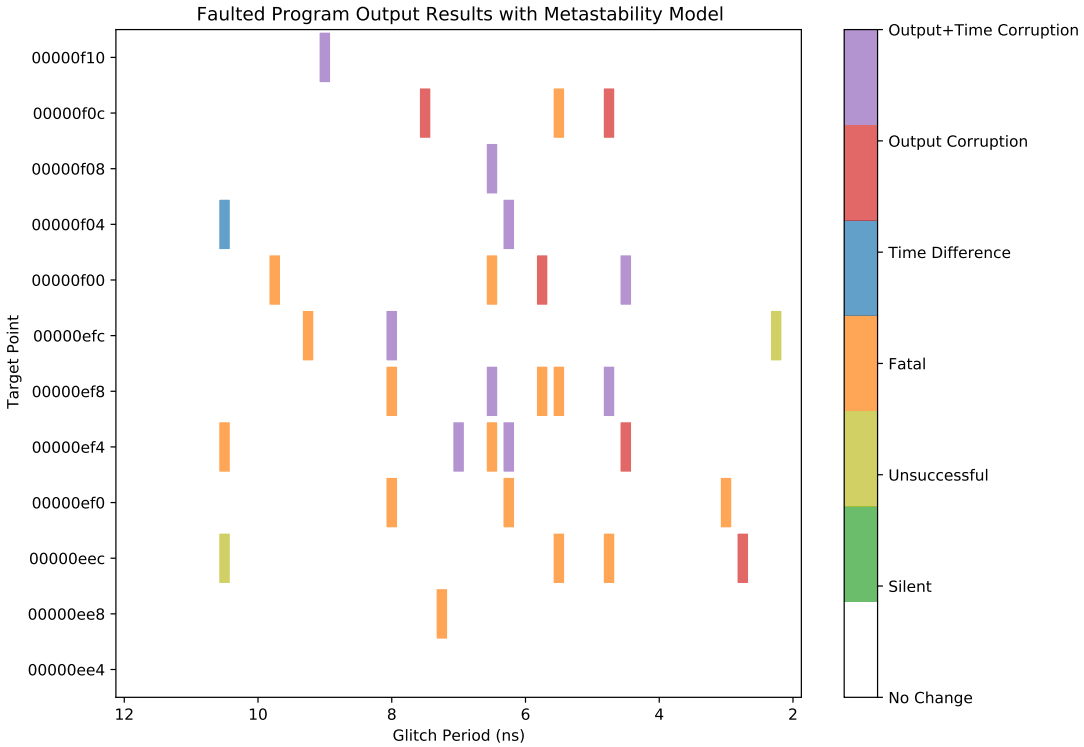


Figure 5.13: Difference in program impacts of glitch attacks between standard simulation and metastability simulation for fault injection on the starting instructions of AES round 1.

5.4 Summary

The two types of tests discussed in this chapter serve different important purposes. First, instruction-level fault response characterization can aid both hardware and software designers. The characterization results reveal how the hardware is vulnerable to fault attacks, and if there are any unbalances in which components experience more severe fault-induced errors. For example, the results explored in Section 5.1.2 showed that various source registers contribute differently to overall fault response, and even that the source 1 and source 2 operand datapaths are faulted in different ways. These results inform software designers whether certain processor components are more prone to successful fault injection so they can take preventative measures when developing critical code.

Second, application-wide fault evaluation aids software designers by analyzing which fault/instruction pairs result in various negative program outcomes. This detailed information provides a method for deeply exploring the impacts of fault injection on critical points in an application. As discussed in Section 5.2.2, this type of testing can reveal which points of interest for fault attacks are more vulnerable to glitches, and informs the developer where to focus countermeasure efforts. Although it is not explored in the results discussed here, the hardware fault propagation is captured for each fault trial. This information could be used to further analyze if certain parts of the hardware are more likely to contribute to faulty program outputs. The high-level data explored in Section 5.2.2 shows that certain faults always lead to corrupted program outputs. These injection locations and parameters can be used as starting points to run deeper analysis on the collected hardware data.

Finally, adding the metastability model to SimpliFI evaluation yielded extra information about the processor's fault response not uncovered by the original tests. While the metastability model is not perfectly accurate to real-world flip flop behavior, randomness is present in noise and real-world metastability and adds to the circuitry's fault response. By including a basic approximation of random behavior occurring at a realistic time (ie. setup violations), additional unforeseen fault vulnerabilities can be addressed. When adding metastability modeling to instruction sequence tests, further microarchitectural fault responses are revealed, such as increased bit errors in data registers. When added to application-level tests, the metastability model provides a first-level approximation to how much the faulty program outputs will vary due to randomized physical phenomena.

Chapter 6

Conclusion

With extensive research into fault attacks over the years, embedded security researchers and analysts have a strong collective knowledge of what fault attacks are capable of, and how they occur. We are at a point where this vast amount of knowledge can be integrated into automated techniques for hardware and software fault evaluation, and this thesis demonstrates that with the introduction of SimpliFI. Even with a simple injection mechanism, SimpliFI reveals device-specific microarchitectural effects on software fault vulnerabilities. Careful design at the hardware level may be able to mitigate these instruction-dependent vulnerabilities for clock glitch faults; such changes may also consequentially dampen the effects of other fault injection mechanisms as well. As discussed earlier, voltage faults can be emulated with the SimpliFI framework due to timing violation similarities between voltage and clock attacks. While integrating voltage faults into the framework would be an improvement, adding the ability to simulate EM faults would greatly increase the power of SimpliFI. Since EM faults can be considered as sampling faults, the metastability simulation feature of SimpliFI could be a key component for implementing simulated EM faults.

With regard to software-level analysis, the BRISC-V processor evaluation presented in this thesis contains an overwhelming amount of data that could be synthesized into insightful information about the software and processor itself. While this was outside the scope of the thesis, developing advanced analytic extensions for data collected with SimpliFI would be another strong improvement. This would enable near-fully-automated characterization of processor and software fault responses. Taking this one step further, the information obtained from such methods could be used to build a highly-detailed, device-specific fault model that plugs into ISA-level simulators like FiSim. With access to so many fault evaluation methods, the embedded security community would benefit from studies that integrate tools together into powerful, full-stack fault analysis toolchains.

Bibliography

[1] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, 20191213, RISC-V Foundation, December 2019.

[2] "The History of Cybersecurity," CompTIA. [Online]. Available: https://www.futureoftech.org/cybersecurity/2-history-of-cybersecurity/

[3] Boneh, Dan and Demillo, Richard A. and Lipton, Richard J., "On the Importance of Checking Cryptographic Protocols for Faults," in *Advances in Cryptology — EUROCRYPT '97*, vol. 14. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 37–51.

[4] Biham, Eli and Shamir, Adi, "Differential Fault Analysis of Secret Key Cryptosystems," in *Advances in Cryptology — CRYPTO '97*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 513–525.

[5] V. Arribas, F. Wegener, A. Moradi, and S. Nikova, "Cryptographic Fault Diagnosis using VerFI," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 229–240.

[6] B. Yuce, N. F. Ghalaty, and P. Schaumont, "TVVF: Estimating the Vulnerability of Hardware Cryptosystems against Timing Violation Attacks," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2015, pp. 72–77.

[7] Riscure, "Riscure FiSim," GitHub, 2020. [Online]. Available: https://github.com/Riscure/FiSim

[8] S. Bandara, A. Ehret, D. Kava, and M. Kinsy, "BRISC-V: An Open-Source Architecture Design Space Exploration Toolbox," in *The 27th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, ser. FPGA '19. New York, NY, USA: ACM, 2019.

[9] R. Agrawal, S. Bandara, M. Isakov, M. Mark, and M. Kinsy, "The BRISC-V Platform: A Practical Teaching Approach for Computer Architecture," in *Workshop on Computer Architecture Education (WCAE)*, 2019.

[10] S. Bandara, A. Ehret, D. Kava, and M. A. Kinsy, "BRISC-V: Open Source Architectural Design Space Exploration Toolbox," Tech. Rep. 0V1, October 2018.

[11] T. J. Gabara, G. J. Cyr, and C. E. Stroud, "Metastability of CMOS master/slave flip-flops," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 10, pp. 734–740, 1992.

[12] L. Kleeman and A. Cantoni, "Metastable Behavior in Digital Systems," *IEEE Design & Test of Computers*, vol. 4, no. 6, pp. 4–19, 1987.

[13] J. U. Horstmann, H. W. Eichel, and R. L. Coates, "Metastability Behavior of CMOS ASIC flip-flops in Theory and Test," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 1, pp. 146–157, 1989.

[14] G. F. Chard, O. Koyuncu, T.-P. R. Koh, and S. Dondershine, "Modeling metastability in circuit design," U.S. Patent US7 139 988B2, 2004.

[15] *ARMv7-M Architecture Reference Manual*, 0402E.e, ARM, Cambridge, England, 2021.

[16] *Atmel AVR 8-bit Instruction Set*, 0856J, Atmel Corporation, July 2014.

[17] J. Richter-Brockmann, P. Sasdrich, and T. Güneysu, "Revisiting Fault Adversary Models - Hardware Faults in Theory and Practice," Cryptology ePrint Archive, Report 2021/296, 2021, https://eprint.iacr.org/2021/296.

[18] M. Dumont, M. Lisart, and P. Maurine, "Modeling and Simulating Electromagnetic Fault Injection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 4, pp. 680–693, 2021.

[19] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[20] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[21] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 29–40.

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

[23] J. Proy, K. Heydemann, A. Berzati, F. Majéric, and A. Cohen, "A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures: A Secure Software Perspective," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ser. ARES '19. New York, NY, USA: Association for Computing Machinery, 2019.

[24] T. Trouchkine, "SoC Physical Security Evaluation," Ph.D. dissertation, Université Grenobles Alpes, 2016.

[25] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software Fault Resistance is Futile: Effective Single-Glitch Attacks," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016, pp. 47–58.

[26] B. Yuce, N. F. Ghalaty, C. Deshpande, H. Santapuri, C. Patrick, L. Nazhandali, and P. Schaumont, "Analyzing the Fault Injection Sensitivity of Secure Embedded Software," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 4, July 2017.

[27] *7 Series FPGAs Memory Resources*, 1.14, Xilinx, 2019.

[28] *ModelSim User's Manual*, 10.1c, Mentor Graphics, 2012.

[29] *Vivado Design Suite UserGuide: Logic Simulation*, 2020.2, Xilinx, 2020.

[30] J. Blömer and J.-P. Seifert, "Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)," in *Financial Cryptography*, R. N. Wright, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 162–181.

[31] A. Moradi, M. T. M. Shalmani, and M. Salmasizadeh, "A Generalized Method of Differential Fault Attack Against AES Cryptosystem," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, L. Goubin and M. Matsui, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 91–100.

[32] NIST, "Lightweight Cryptography Benchmarking," GitHub, 2021. [Online]. Available: https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking