A Fault Injection Simulator for ARM

WPI

A Major Qualifying Project
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfilment of the requirements for the
degree of *Bachelor of Science*

Team Members:
Jonathan Metzger
Maryann O'Connell
Himanjal Sharma

Date:
2 March 2018

Sponsors:
Vincent Chen
Andrew Tran

Advisor:
Mark Claypool

School:
Worcester Polytechnic Institute

Table of Contents

Abstract

Fault injection attacks pose a vulnerability to software integrity. A successful attack can result in skipped instructions, introduction of new instructions, and other undesirable behavior. Testing for fault injection vulnerabilities is a common practice when creating systems with a hardware component. Injecting faults enables developers to analyze the effectiveness of preventative measures, ultimately leading to improved hardware tolerance and overall application dependency. Testing can be achieved using hardware fault injections or software fault simulations. Hardware fault injection is expensive and difficult to scale compared to software solutions. Using a software simulation approach adds flexibility, protects hardware and simplifies testing procedures. Software simulation utilizes a virtual development kit (VDK) to simulate hardware components and injects faults into it.

Our goal is to study the behavior of voltage fault injection in hardware and design an application to enable developers to inject fault-inducing instructions at user-selected trigger points within the source code. In order to prevent hardware destruction and verify system dependability, we simulated fault behavior in a virtual environment with ARM7 architecture. Faults are simulated through a GDB server to assess the effectiveness of preventive measures implemented at the software level. We successfully built a fault injection simulator prototype based on fault behavior we observed. Our goal is to provide a data driven simulation combining the results from hardware testing and software fault injection testing to increase flexibility and improve accuracy of software testing.

1. Introduction

Hardware fault injection was first introduced in the 1970s as a technique to improve the dependability of systems (Carreira et al., 1999). One fault injection technique is to use pin-level probes to apply a voltage to a system at the boundary of safe operation (Hsueh et al. 1997). This technique can result in damaged hardware, which results in increased testing expenses. Currently, the software security team at Nvidia uses voltage hardware fault injection to test products prior to release. They are working towards an alternate approach to test system on chips (SoCs) against fault injection by simulating faults in Virtual Development Kits. Our project aimed to understand how fault injection impacts hardware and create a graphical user interface (GUI) application with a command line interface (CLI) that simulates hardware fault injections via a GDB server to emulate faults in Nvidia systems on a chip (SoCs).

The two components of our project included a hardware study and fault injection simulator. We studied how registers respond to voltage fault injection in hardware and built an application that modifies register values using GDB, the GNU Project debugger (GNU, 2017). To provide a better user experience, we built a GDB Client application and interface. The interface supports fault injection simulation and enables users to select a trigger point within the target source code. A high level overview of our project architecture is illustrated in Figure 1.

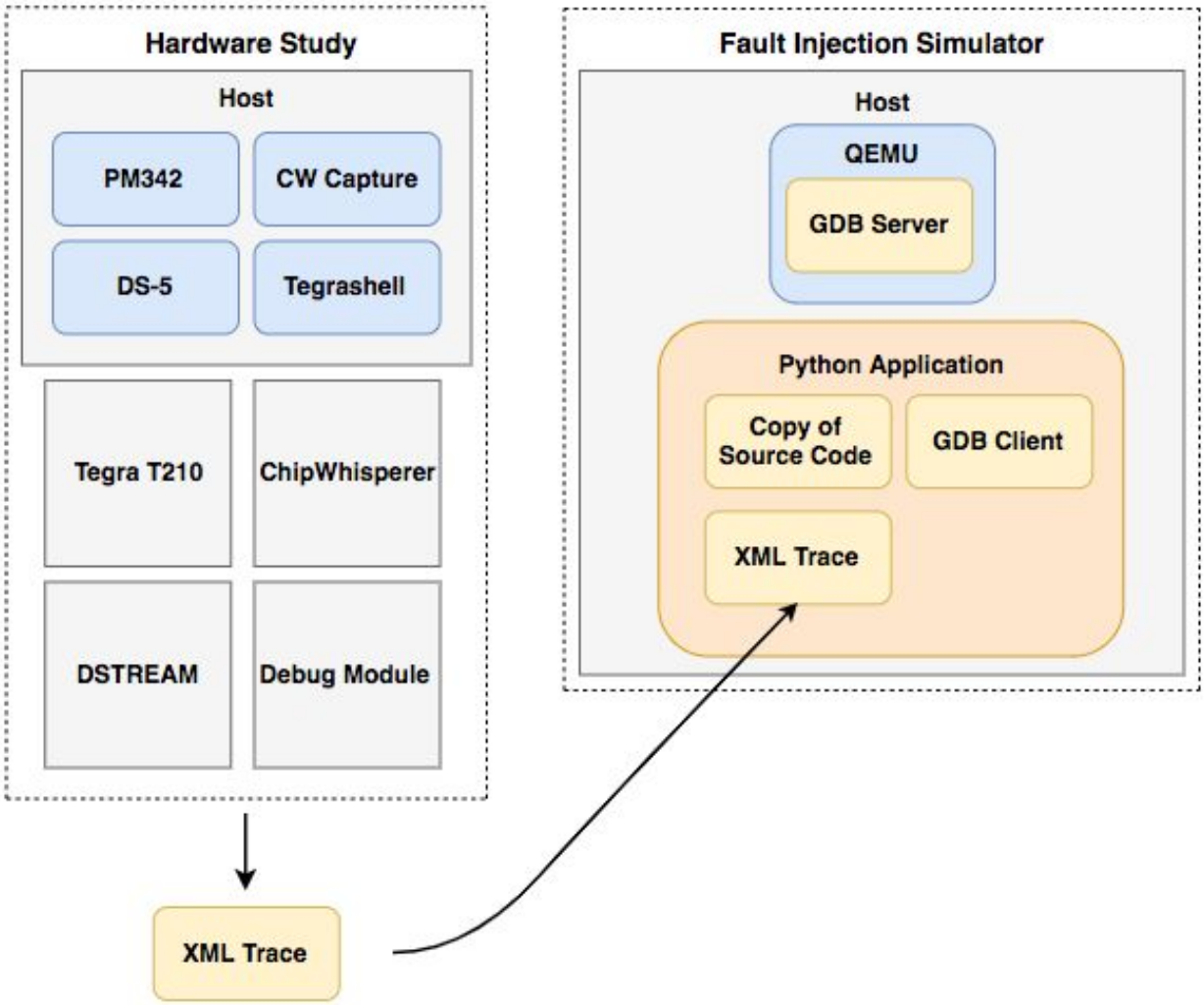

Figure 1. Project Architecture Overview.

The hardware study involved writing a set of scripts and running a variety of software to trigger voltage fault injection in a Tegra T210 target. The fault injection simulator was a Python application built to connect to Qemu through a gdb server. The hardware data was loaded in an XML document and imported into the simulator to replicate glitch behaviour on source code.

The results from hardware study suggest that load instructions are most susceptible to voltage fault injection whereas branching instructions are least susceptible to voltage fault injection. The results also suggest move instructions are vulnerable to voltage fault injection, and that negative values may increase the chance of unexpected software behavior resulting from fault injection.

The fault injector simulator prototype was finished and tested on simple C program by injecting faults into the program and exiting it abnormally. The simulator emulates faults by changing register values and editing program behaviour. The simulator is capable of triggering a fault at a user defined point in the program and can also execute a list of faults from a XML file.

Chapter 2 briefly explains fault injection and its consequences. It also describes the components and technologies used in our hardware study and during software development.

Chapter 3 describes the methods we used to study the effects of voltage spikes on hardware and to develop the fault injection simulator. It also shows how these objectives intersect.

Chapter 4 explains hardware study setup and steps taken to glitch the hardware and monitor the effects. It also shows data collected and results of study.

Chapter 5 describes the design process of the application. It also describes each component of the interface and explains the implementation process.

Chapter 6 concludes our project and describes the future steps that could be taken by future developers.

2. Background

This chapter summarizes relevant fault injection research. It provides an explanation of hardware and software fault injection, as well as consequences associated with fault injection attacks. This chapter includes information fundamental to our hardware study and simulator development, including the devices and applications used to construct our tool.

2.1 Fault Injection

Fault injection testing evaluates the dependability of computer systems, hardware and software, through various methods in order to test fault-tolerant systems or components via fault detection, isolation, reconfiguration and recovery capabilities (Hsueh et al., 1997). Testing for fault injections, requires an understanding of the system's architecture, structure and behavior, tolerance for faults and failures, built-in detection, and recovery mechanisms. A solid understanding of these components is especially critical for assessing larger systems. Once the system is understood, specific instruments and tools to inject faults are leveraged to create measurable faults. The ideal fault injection technique is highly dependent on specific product needs.

Hardware and software fault injections are differentiated by the type of fault being created. For example, "stuck-at faults"—faults that force a permanent value into a point in a circuit—require a hardware injector in order to control the location of the fault (Kooli and Natale, 2014). Corrupting data requires a software injector.

Bit-flips, on the other hand, are attainable through both hardware and software fault injections.

2.1.2 Hardware Fault Injection

Hardware fault injection requires close proximity to the target device, but can be achieved with or without contact. To inject a fault, without contact, physical phenomena such as heavy radiation or electromagnetic interference are used (Hsueh et al., 1997). Voltage fault injection is an example of a method requiring direct contact. Probes are attached to pins and voltage to the target is cut off or increased. The exact voltage is fine tuned to find the boundary of safe operating. Too little power results in the device turning off and too much power can permanently damage the device therefore determining this range is a critical component. The risk of damaging the target is higher with hardware fault injection and therefore generally costs more than software fault injection. A fault that results in unexpected software behavior is commonly referred to as a glitch.

2.1.3 Software Fault Injection

Software fault injection provides an inexpensive, scalable way to test for fault injection. Faults can be injected into software at either compile-time or runtime (Hsueh et al. 1997). In order to inject a fault at compile-time, the program instruction must be modified before the program is executed. This will inject errors into source or machine code to emulate faults. The code modifies the target

program causing injection. Runtime injections are triggered using time-outs, exception handling code, or code insertion (Hsueh et al. 1997).

2.1.4 Consequences

Embedded systems without logical exploitable code may still be exploited through fault injections. Instruction corruption and instruction skipping are possible when a variable number of bits are flipped due to fault injection (Timmers et al., 2016). Instruction corruption describes the behavior occurring when the original instructions are modified to architecture supported instructions. Instruction skipping entails skipping instructions that do not impact the state or result in the introduction of new instructions.

An article from 2011 describes a group of attacks that claimed to have beaten the Xbox 360 security ("Hackers Claim to Have Beaten Xbox 360 Security", 2011). A released a video showed how hackers succeeded in bypassing the security system during runtime, targeting the CPU on Microsoft's Xbox 360 (YouTube, 2011). The video showed how slowing the CPU speed during the boot sequence enabled reset of a circuit at a specific time point. This caused the Xbox to fail to properly check the bootloader signature, allowing hackers to run their own bootloader. This bootloader can include malware that can affect performance or steal sensitive information.

2.1.5 Fault Injection and ARM

Recent research publications explore the correlation between faults and specific ARM instructions (Timmers et al., 2016). ARM load (LDR/LDMIA) and store (STR/STMIA) instructions are vulnerable to attack as they are not fault injection secured . The chance of success is improved because LDR and STR operations are executed multiple times during the startup process and operations are performed on attacker controlled data. An attacker can load a value into the program counter (PC) through a single or double bit corruption. "Setting the PC as the destination for the LDR or LDMIA" is often the initial step to later achieve arbitrary code execution (Timmers et al., 2016). A secure boot attack can be conducted by identifying the destination address of the copy instruction, overwriting flash storage with a malicious payload, then injecting a fault after the shellcode is copied into internal memory.

On the contrary, branching instructions are reportedly not vulnerable to errors caused by voltage glitching (Barenghi et al., 2009). CPU registers are designed with a "low capacitance" and an "instruction buffer" separates the CPU and memory, "which cuts down the capacitive load of the path to the instruction cache and the program memory" (Rivière et al., 2015). These findings are fundamental to assessing the validity of our results.

2.1.6 Fault Injection Summary

Our project is concerned with threats posed by fault injection attacks and focuses on providing a data driven approach for identifying fault vulnerabilities to reduce the risk of consequences, such as those described in section 2.1.4. Fault behavior is studied using hardware fault injection, specifically, voltage fault injection. The observed behavior is then imitated using simulated faults. The targets in our project use an ARM7 architecture and the simulator tool built supports ARM7.

2.2 Hardware Study Components

Our hardware study leveraged both Nvidia and third-party hardware and software tools. This section provides background for the hardware devices used during the study, including Tegra T210, ChipWhisperer, and DSTREAM (see Fig. 2). It also introduces the software used in this study, including Tegrashell, CWCapture, and DS-5.

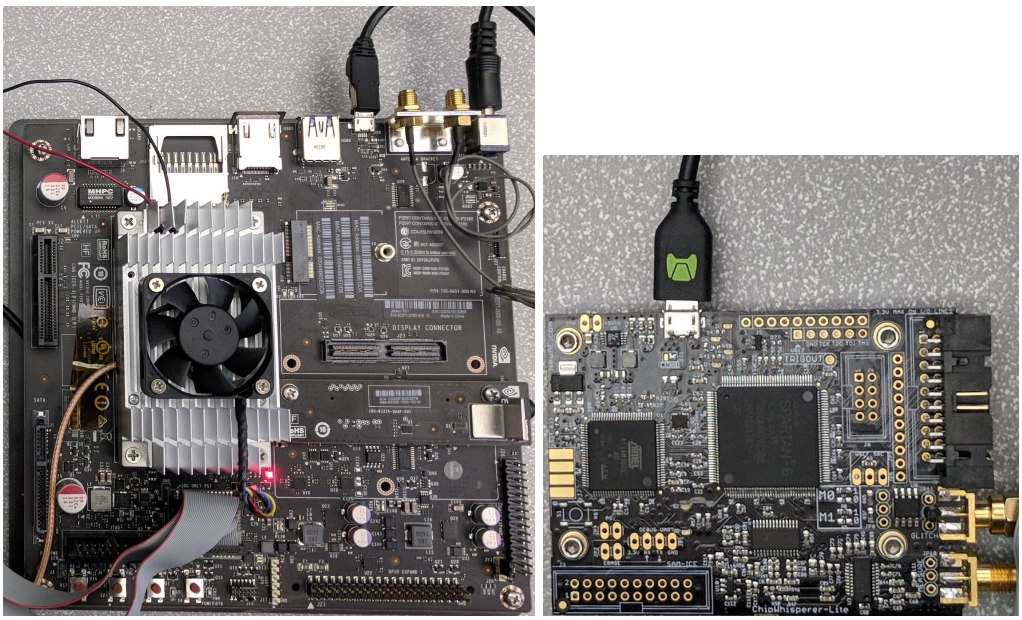

Figure 2. Hardware study devices.
Tegra T210 (left) and ChipWhisperer-Lite (right).

2.2.1 Tegra and Tegrashell

Nvidia develops hardware including the system on a chip (SoC) series, Tegra. Our study utilized the Tegra T210, also known as Tegra X1, which is used in the Nintendo Switch. Tegra T210 uses an ARM7 processor and provides a JTAG interface which can be used for programming the T210 and debugging.

Tegrashell is a field-programmable gate array (FPGA) debugger tool that includes the ability to dump register values (see Fig. 3). Used together, the Tegra T210 and Tegrashell provide a useful set of tools for testing and logging.

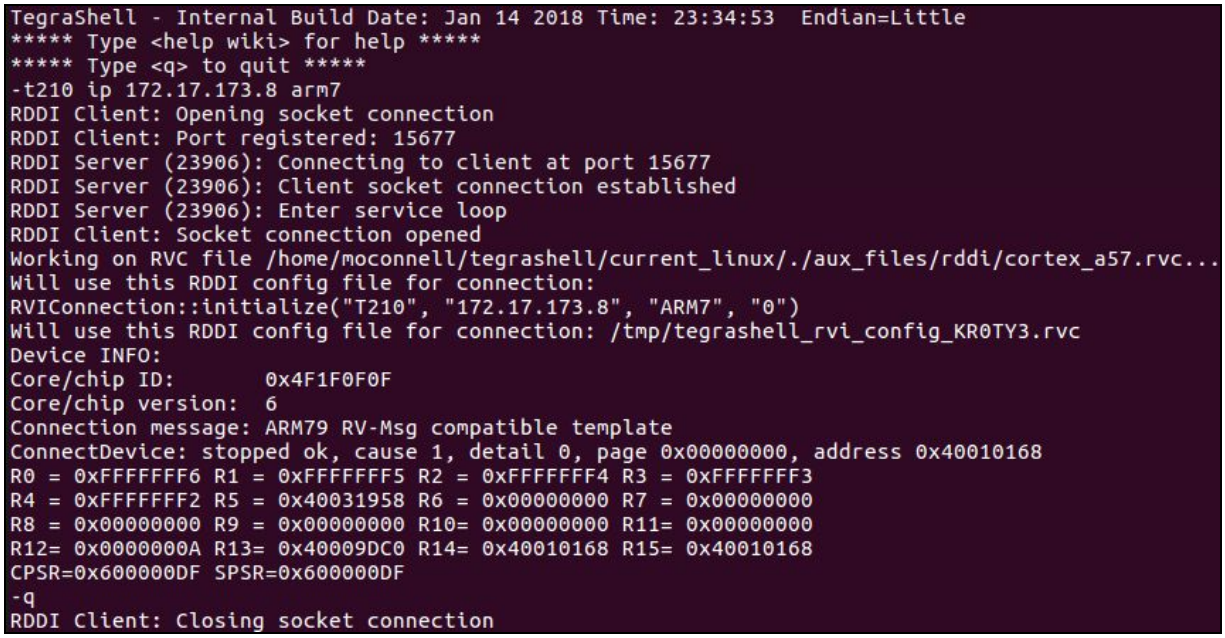

Figure 3. Tegrashell output for a single sample.

2.2.2 ChipWhisperer and CW Capture

ChipWhisperer (CW), made by NewAE Technology Inc., is a tool designed for embedded hardware security research. We used the ChipWhisperer Lite (CW1173) board to glitch a Tegra target. The CW1173 provides the option to trigger clock generated voltage glitches using a built-in electronic switch (MOSFET). The MOSFET shorts the power to ground during low-power glitches and increases the power during high-power glitches. CW Capture is a software program used to control a ChipWhisperer board (see Fig. 4). Glitch parameters, including high-power and low-power glitching, glitch repeat, and clock frequency, are set using CW Capture.

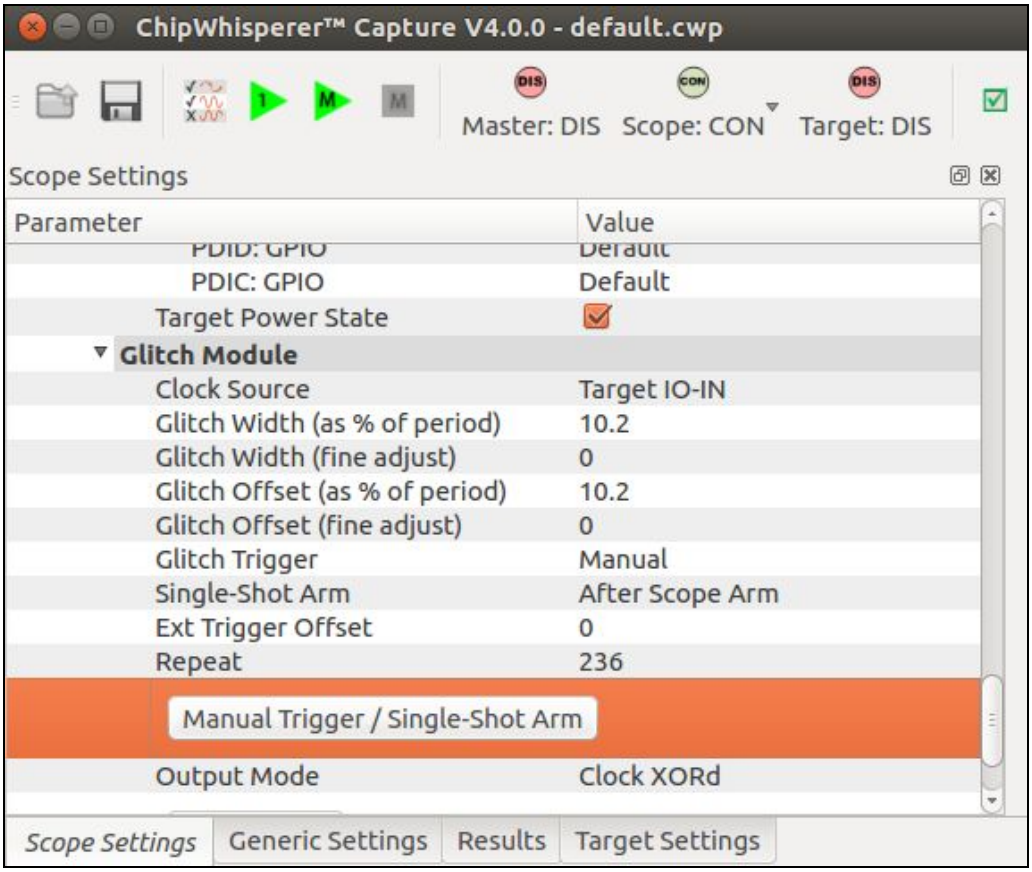

Figure 4. GUI-version of CW Capture software highlighting.

2.2.3 DSTREAM and DS-5

DSTREAM is a high-performance debug and trace unit created by ARM which supports ARM architecture versions v4 through v8 (Arm Holdings, 2018). ARM Development Studio 5 (DS-5) is a set of software tools which includes an Eclipse IDE, ARM compilers, and an advanced debugger. Nvidia has a custom configuration available for connecting Tegra T210 using DS-5. DSTREAM and DS-5 are beneficial for manual testing and viewing register changes when glitching manually.

2.2.4 Other Components

A Nvidia T114 Debug Module was used with PM342 software to automate physical resets of the Tegra T210 (see Fig. 5). This is a convenient alternative to manually pushing the reset button. The T114 Debug Module was also used to connect DSTREAM for debugging and verification purposes.

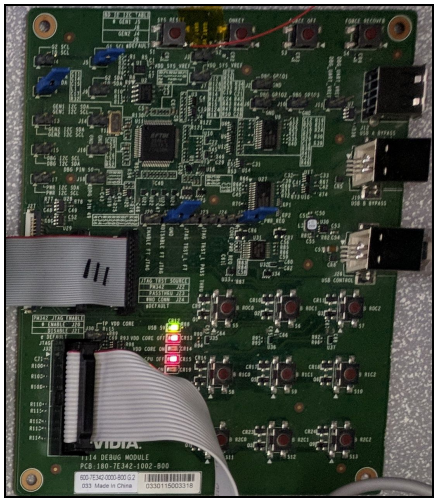

Figure 5. T114 Debug Module.

2.3 Simulator Application Components

This section describes third party applications critical to the development of our application, including GDB, the GNU Debugger and Qemu. This section also covers Synopsys VDK, a technology crucial to future work related to our tool.

2.3.1 GDB

We used `arm-none-eabi-gdb` debugger from Arm Toolchain for Ubuntu. The toolchain provides the ability to remotely send and receive program data. The GDB client was executed as a subprocess from Python and all the input and output was piped to the command line. Once launched, we were able to attach the client process to the debugging server running on Qemu virtualizer.

2.3.2 Qemu

Qemu is a generic and open source machine simulator and virtualizer (Qemu, 2018). A developer can use the full-system simulator, user-mode simulator or a virtualization tool. We used Qemu for its simplicity and its ability to virtualize an entire operating system to test our glitches.

2.3.3 Synopsys VDK

Virtualizer Development Kits (VDKs) are software development kits containing design-specific virtual prototypes as well as debug and analysis tools and sample software. Synopsys supplies custom VDK solutions, including VDKs used to simulate embedded systems. Synopsys' VDKs interface seamlessly with DS-5, creating an integrated debug flow with extended control and analysis capabilities (Synopsys, 2017).

3. Methodology

Our goal was to assist Nvidia with understanding and testing for fault injection. Our project consisted of two related objectives:

1. Understand the effect of fault injection on hardware.
2. Build an application to simulate fault injection behavior.

Our project focused specifically on voltage fault injection due to recent voltage fault injection attacks against Nvidia SoCs. We created a working simulator to help Nvidia engineers test their source code for fault vulnerabilities. The two objectives intersect to provide a mechanism for Nvidia to identify vulnerable source code. Figure 6 illustrates how our objectives converge to provide Nvidia with a process for mitigating faults at the software level.

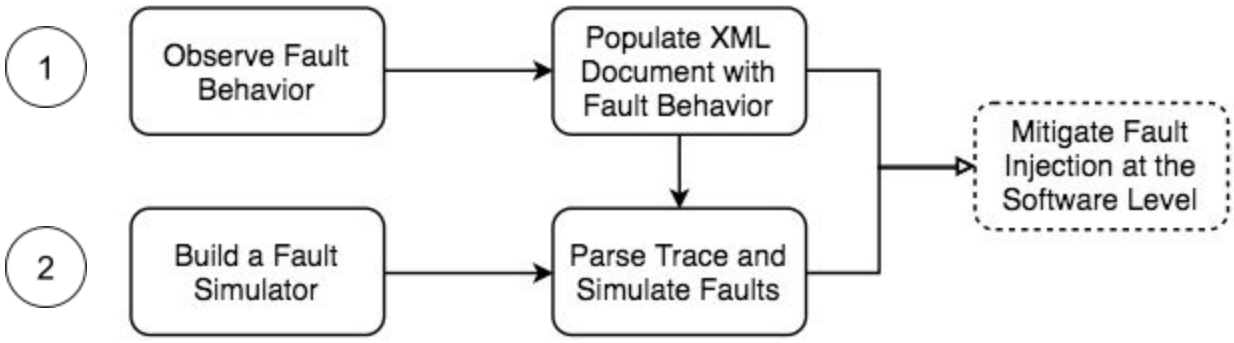

Figure 6. Methodology overview—the first level depicts the tasks to achieve our first objective, and the second level illustrates the tasks for our second objective. Arrows indicate the order of task completion, therefore the initial tasks for both objective were completed simultaneously. Dotted lines denote related tasks outside the scope of our project.

To work towards both objectives, our team met with our mentor and manager on a semi-weekly basis. Each meeting we established intermediate goals

that we completed. This practice helped us iterate over multiple application prototypes and refine our hardware study. A detailed explanation of how we collected data for our hardware study and how we built our simulator is provided in Chapters 4 and 5.

4. Hardware Study

The hardware fault injection study was designed to observe the impact of fault injection on Nvidia Tegra systems. This chapter outlines the settings and methodology used for our hardware study. It concludes with a discussion of our results.

4.1 Hardware Study Experimental Settings and Methodology

Our hardware study was designed to systematically collect and analyze behavior in response to voltage fault injection. The design of our study was influenced by hardware testing previously completed at Nvidia. Past testing included the use of third-party hardware components such as a ChipWhisperer and a DSTREAM device. Prior work influenced voltage fault injection parameter selection and initial test code development as well.

4.1.1 Experimental Setup

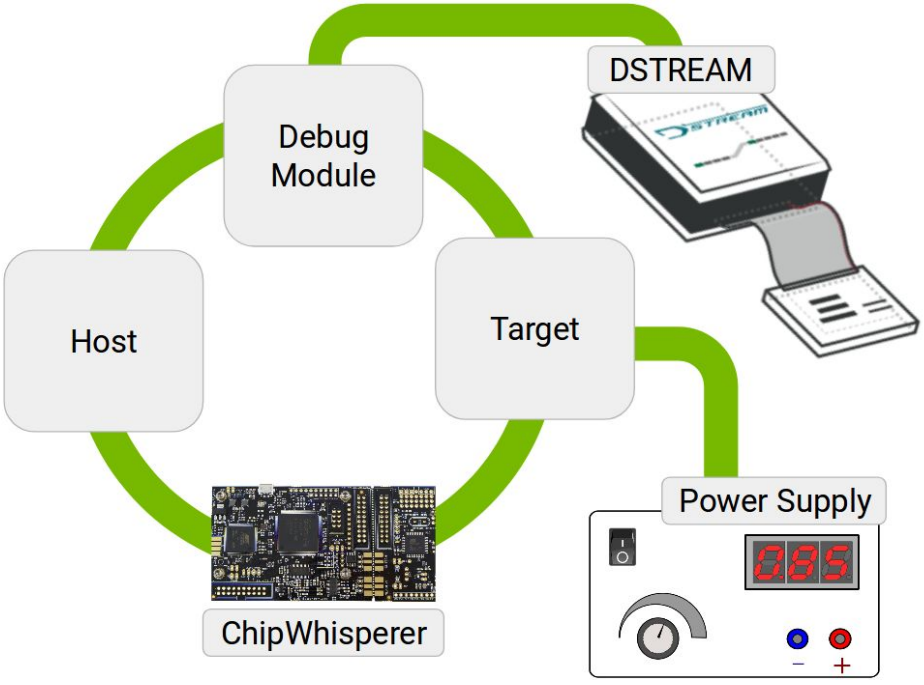

Figure 7. Overview of hardware study components.

Before collecting data, we setup the hardware components required for our study (see Fig. 7). Throughout our experiment, our target was powered by a Hewlett-Packard E3632A DC power supply. The Tegra T210 target was connected to our host machine via a micro USB cable, to a ChipWhisperer-Lite via a SMA cable attached to the glitch port of both devices, and a Nvidia T114 debug module via a custom breakout board connector. The ChipWhisperer was attached to and power by our host via a micro USB cable. The debug module also connected to our host machine via USB-B and a DSTREAM device via a 20-pin ribbon cable. The DSTREAM device was hooked up to the Internet and we were able to mount the device by its IP address using DS-5. We ran a set of Bash and Python scripts on our host to

configure the ChipWhisperer through CWCapture. To collect 1000 samples for each test, we looped through instructions to reset the target using the debug module, trigger a voltage glitch, and log target register values with Tegrashell.

4.1.2 Voltage Fault Injection Parameters Selection

Using DS-5 and CWCapture, we determined the ideal voltage supply and ChipWhisperer parameters to use throughout our experiment. Powering the target with a low voltage supply makes the target more susceptible to voltage glitches. However, if the power supply is too low, DS-5 cannot connect to the target. Through trial and error, we identified 0.851V as the minimum voltage supply necessary to power a Tegra T210 during our experiment.

For each test, CWCapture was launched using a Python script which enabled us to set custom parameters. The ideal glitch parameters reliably glitch the target in as few clock cycles as possible. Based on previous work completed at Nvidia with a ChipWhisperer, we selected 204 MHz for the desired clock frequency. We used the default glitch width and default glitch offset from the clock edge, both 10.2% of the ChipWhisperer clock period. The repeat parameter in the CWCapture glitch module reflects the number of ChipWhisperer clock cycles to glitch the target. Using CWCapture, we tried various repeat values and manually glitched the target, then verified register values with DS-5. We fine tuned this parameter to only skip one instruction. Ultimately we selected a repeat value of 236 that glitched the target

approximately 25% of the time when looping through copy instructions. Both high power and low power glitches were used for the duration of the experiment.

4.1.3 Test Codes

Eight inline assembly test codes were added to the cold boot firmware for Tegra T210 and flashed in order to observe the impact of fault injection. All tests were designed to utilize different types of assembly instructions. Additionally, the first three pairs of tests were intended to detect the significance of positive and negative values. A combination of looping and branching the current instruction were used to compare outcomes associated with these instruction types and interrupt normal program execution. The last two tests were designed to increase the chance of detecting when a glitch occurred. Test 1 copied the values ten to fourteen to registers R0 through R4, respectively, within an infinite loop.

```
...
asm volatile("loop:"
             "mov r0, #10 \n\t"
             "mov r1, #11 \n\t"
             "mov r2, #12 \n\t"
             "mov r3, #13 \n\t"
             "mov r4, #14 \n\t"
             "b loop"
);
...
```

Listing 1. Test 1: Copy Small Constants Loop Inline Assembly Code

Negative values were substituted into the move instructions used in test 1 to create test 2. Test 3 loaded 0x55555555 into registers R0 through R4, setting odd bits to one within an infinite loop. Test 4 mirrored test 3, setting even bits to one.

Test 5 copied the values ten to fourteen to registers R0 through R4, respectively, then branched at the current instruction. Similar to test 2, negative values were substituted for positive values in test 5 to form test 6. Test 7 copied zero to registers R0 through R4 and incremented the values in an infinite loop. Test 8 alternated loading zero and 0xAAAAAAAA into registers R0 through R4 within an infinite loop. All test codes are included in Appendix A.

4.1.4 Data Collection and Log Analysis

Data collection took place during each test. Python subprocess calls were appended to the CWCapture Python script to execute additional Bash scripts and terminal commands. Adding to the CWCapture script helped us improve test performance by performing the initial CWCapture setup just once per a test code. Subproccesses were used to call programs to reset the target and dump register values for each sample. After flashing the modified cold boot firmware to the target, we ran a Bash script used to launch the CWCapture Python script and parse program output. The output from Tegrashell register dumps was piped to a log file for processing. The parsed log file was saved as a CSV file and imported to Google Sheets for analysis.

Google Sheets was the primary tool used for processing the raw register data from each test. Using conditional formatting and a Google Apps Script, we identified glitched samples. Enumerated register value tables were created for tests with a finite number of expected register values where at least one glitch occurred.

Disassembly code for each expected instruction and instructions resulting from a glitch was obtained using DS-5 and summarized in table format. Frequency glitch patterns were identified for analysis.

4.1.5 Assessments

During the experimental setup, manual verification was used to confirm that glitch parameters and tests codes were reasonable, and that all hardware was working as expected. Study assessments took place subsequently after each test was completed and results were analyzed. The process of repeat evaluations helped us refine our study and write tests to cover greater depth and breadth. Substantial troubleshooting was required during early configurations because we were inexperienced in the technologies used and determining appropriate experimental parameters required considerable attention. After the setup was finished, we originally completed the study for three test codes (tests 1, 4, and 5). Discussion with the Nvidia team prompted us to design two additional tests (test 7 and 8) to increase the visibility of glitches by narrowing the expected register values to one address. Table 1 illustrates how alternating register values in Test 8 enabled us to spot skipped instructions that we may have missed in Test 1.

Table 1
(a) Example of undetectable skipped instructions in Test 1.

	R0	R1	R2	R3	R4	PC
	#0xA	#0xB	#0xC	#0xD	#0xE	0x40010168
skipped ↓	~~#0xA~~	~~#0xB~~	~~#0xC~~	~~#0xD~~	~~#0xE~~	~~0x4001016C~~
	#0xA	#0xB	#0xC	#0xD	#0xE	0x40010170

(b) Example of increased visibility of skipped instructions in Test 8.

	R0	R1	R2	R3	R4	PC
	#0xA	#0x0	#0x0	#0x0	#0x0	0x40010180
skipped ↓	~~#0xA~~	~~#0xA~~	~~#0x0~~	~~#0x0~~	~~#0x0~~	~~0x40010184~~
	#0xA	#0x0	#0xA	#0x0	#0x0	0x40010188

The final study modification led us to add tests to examine the relationship between positive and negative values on fault injection vulnerability. We created tests to complement test 1, 4, and 5.

4.2 Hardware Study Results

After collecting and summarizing our study data into tables, we looked for patterns and observable behavior. Patterns were detected by first enumerating the glitched register values then spot checking these values. We primarily focused on three metrics:

1. Number of glitched samples per test.
2. Unexpected values assumed by registers R0-R4.

3. Unexpected values assumed by the program counter.

These metrics were intended to help characterize the vulnerability of test codes to voltage fault injection. This section reports significant findings, including the effect of positive versus negative values and the impact of specific instructions on likelihood of glitching. Additional result tables can be found in Appendix A.

4.2.1 Instruction Vulnerability

Table 2
Test code reference.

Test #	Test Description	Other
1	Copy small constants and loop	Positive values
2		Negative values
3	Load large constants and loop	Positive values
4		Negative values
5	Copy small values and branch current instruction	Positive values
6		Negative values
7	Copy zero sequence and increment by 1 loop	Increased glitch visibility
8	Load alternating small constants and loop	Increased glitch visibility

Table 2 is included above to provide a quick reference to the types of test codes used and their corresponding number. Tests 1, 2, 7, and 8 revealed move, move negative, and add instructions are subject to glitching roughly one-quarter of the time with the parameters used (see Table 3).

Table 3
Results Overview for Tests 1, 2, 7 and 8.

	Test 1	**Test 2**	**Test 7**	**Test 8**
Normal Execution	74.4%	62.6%	59.5%	74.2%
Glitch Detected	24.7%	34.9%	28.2%	24.4%
Unknown	0.9%	2.5%	12.3%	1.4%

Unknown samples indicate no register values were obtained for that sample. Test 7 also had a high number of unknown samples, which may have been the result of glitching too hard. Of these tests, test 7 and test 8 had the highest glitch visibility because values for registers R0 through R4 were expected to vary each instruction. Therefore, the rate of glitching may have been higher for test 1 and 2 because there was no way of knowing if the program skipped ahead to an expected instruction without affecting the value of registers R0 through R4.

Table 4 shows the summarized results for tests 3 through 6, inclusive. The results from tests 3 and 4 suggest that load instructions are most susceptible to voltage fault injection. The results from tests 5 and 6 suggested that branching the current instruction is least susceptible to voltage fault injection. Both these findings are consistent with previous research which showed load and store instructions are vulnerable and branching the current instruction is less susceptible (see Background section 2.1.5).

Table 4
Results Overview for Tests 3 through 6.

	Test 3	Test 4	Test 5	Test 6
Normal Execution	35.5%	2.2%	98.6%	95.3%
Glitch Detected	61.9%	96.6%	0%	3.6%
Unknown	2.6%	1.2%	1.4%	1.1%

4.2.2 Positive versus Negative Values

Three sets of tests were conducted with both positive and negative values. The disassembly code shows tests 1 and 5 utilized move instructions, tests 2 and 6 used move negative instructions, and tests 3 and 4 made use of load instructions (see Table 5).

Table 5
Disassembly of Expected Instructions for Tests 1 through 3.

	Disassembly		
PC	**Test 1**	**Test 2**	**Test 3**
0x40010168	MOV r0,#0xa	MVN r0,#0x9	LDR r0,[pc,#1572]
0x4001016C	MOV r1,#0xb	MVN r1,#0xa	LDR r1,[pc,#1568]
0x40010170	MOV r2,#0xc	MVN r2,#0xb	LDR r2,[pc,#1564]
0x40010174	MOV r3,#0xd	MVN r3,#0xc	LDR r3,[pc,#1560]
0x40010178	MOV r4,#0xe	MVN r4,#0xd	LDR r4,[pc,#1556]
0x4001017C	B {pc}-0x14	B {pc}-0x14	B {pc}-0x14

Figure 8 graphically depicts the number of normal samples versus glitched samples for the first three sets of tests. The results of these tests suggests negative

values may be more susceptible to voltage fault injection. More research is required to determine if this finding is significant.

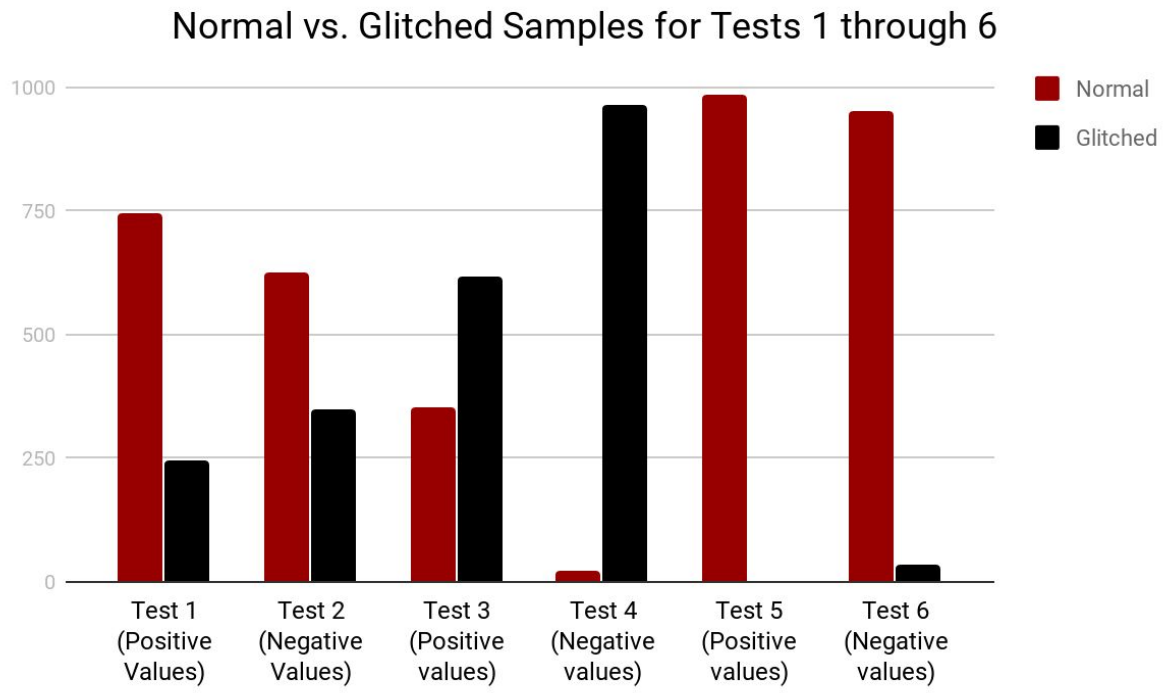

Figure 8. Normal versus Glitched Samples for Tests 1 through 6.

4.2.3 Bit Patterns

We looked for patterns in the glitched values of registers R0 through R4. The majority of glitched values appeared to be some constant set by error handling code. However, a few other distinct patterns emerged. These include adding one, adding another small constant, and flipping the signed bit from zero to one. We observed non-signed bits flipping from both zero to one and one to zero. Table 6 demonstrates some of these patterns.

Table 6
Examples of Distinct Bit Patterns for Glitched Samples. Glitched bits are bolded.

Test #	Register	Expected Value	Actual Value
1	R1	0x0000000B	0x0080000B
		0000 0000 0000 0000 0000 0000 0000 1011	0000 0000 **1**000 0000 0000 0000 0000 1011
1	R4	0x0000000E	0x0000000F
		0000 0000 0000 0000 0000 0000 0000 1110	0000 0000 0000 0000 0000 0000 0000 111**1**
2	R1	0xFFFFFFF5	0xFFFFFFF4
		1111 1111 1111 1111 1111 1111 1111 0101	1111 1111 1111 1111 1111 1111 1111 010**0**
3	R0	0x55555555	0xD5555555
		0101 0101 0101 0101 0101 0101 0101 0101	**1**101 0101 0101 0101 0101 0101 0101 0101

Out of the five registers we modified, R4 was least likely to be impacted by a glitch. The underlying cause for this behavior is unknown, however, it is worth recalling that register R0 through R3 are typically "caller-save registers", meaning they may be changed then later restored by a subroutine (Burch, 2012).

5. Fault Injection Simulator

Hardware fault injection testing can provide useful insights, however, it is an expensive form of testing because it can permanently damage hardware, can be time consuming, and difficult to scale. It also requires physical access to the device, which may not be available during all stages of product development. To solve these issues, we created a tool to test software for fault injection by simulating fault behavior.

Fault injection simulator is an application used to simulate voltage glitches in a virtual environment. In this chapter we discuss the goals set for the application and the design decisions that were made to create the application. Finally, we discuss the implementation process to achieve the goals set for the fault injection simulator.

5.1 Application Design

This section begins by defining the goals of our application to provide context for our design decisions. We describe the environment we worked with and the functionality we sought to provide. We conclude by highlighting the user interface design.

5.1.1 Application Goals

We established application goals through semi-weekly team meetings with our mentor and manager to meet the needs of the Tegra software security team at Nvidia. The application was designed with the following goals in mind:

1. Build a generic application to facilitate testing a variety of software for fault injection vulnerabilities.
2. Provide the ability to simulate fault behavior from hardware traces.
3. Support manually triggering the program and executing batch fault simulations.

These goals formed the foundation for the design decisions described in forthcoming sections.

5.1.2 Virtual Environment

We used Qemu, a generic and open source machine simulator and virtualizer, in order to simulate the system. Qemu can run a variety of operating systems and programs and supports C to ARM cross compilation (Qemu-arm). Once the user imports the source code, ARM gcc compiles it. Qemu allowed our fault injection simulator to connect to a GDB server through a serial port, send commands, and receive data about the program in real time.

5.1.3 Testing Source Code

Users can test source code against voltage glitching using the fault simulator. The fault simulator connects to the source program running in a virtualizer and injects faults while monitoring the program for errors and execution order. Users can inject glitches by manually entering commands and changing program registers from the simulator. A user can also use one of the automated glitching features inbuilt into the simulator. Various ways to interact with the simulator can be seen in Figure 9.

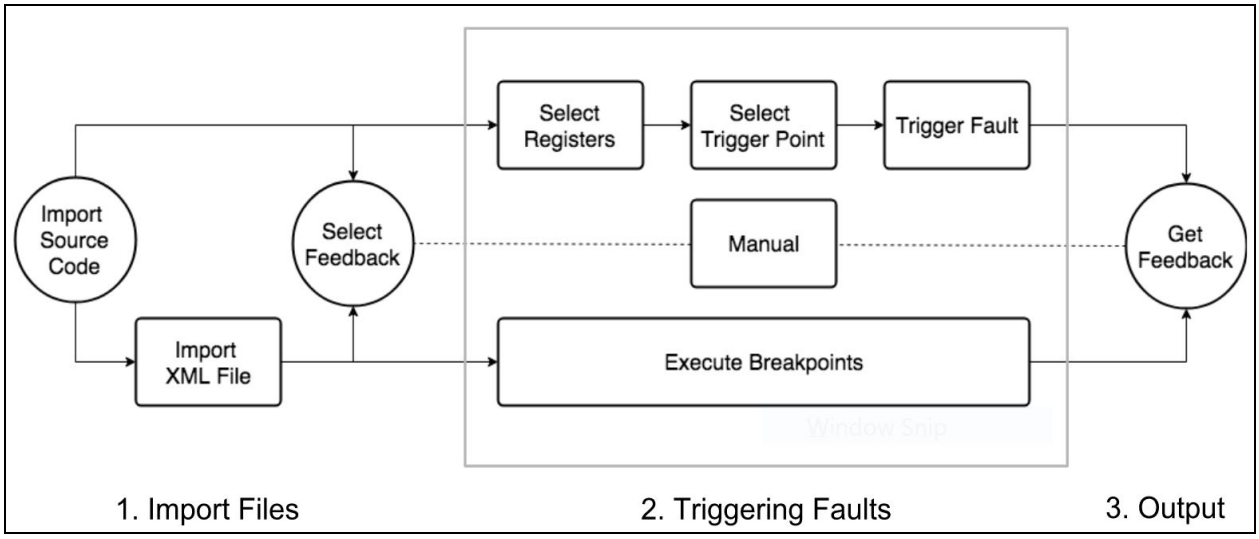

Figure 9. Fault Injection Simulator Flowchart.

5.1.3.1 Trigger Single Fault

In order to trigger a fault at a specific address in the program the users follow the following steps:

1. **Import Source Code:** Import the program file to be tested using the select file dialog box.
2. **Select Feedback:** Update the program feedback address by selecting the program line in the text box and clicking “Update Feedback” Button.
3. **Select Registers:** Select registers to manipulate during fault simulation from the register drop down menu.
4. **Select Trigger Point:** Update the glitch trigger address by selecting the program line in either source code or machine code.
5. **Trigger Fault:** Click “Trigger Fault” to start the test.

Once a test is started the simulator launches the program and halts the execution at the trigger point to change register values. The simulator then updates the value of register using a predefined pattern and continues the program execution. If the program reaches the feedback line, the simulator provides successful feedback to the user. If the program exits abnormally or gives an error before reaching the feedback line, the simulator provides failure feedback to the user.

5.1.3.1 Trigger Multiple Faults

In order to execute multiple faults at once the users must store all the fault data into an XML File. The data includes the trigger address, registers to be manipulated and the pattern to be applied to them. Once all the data is imported into the simulator the user can execute them all by clicking “Execute XML” button. The simulator reads the data from XML and executes each fault on the program running inside the virtualizer. As the test proceeds the simulator provides feedback to the user by displaying green (success) or red (failure) color for each fault.

5.1.4 Feedback and Logging

Prior to starting a test users are prompted to select a feedback line. This step is required to tell the simulator when to stop the running tests. Feedback line is usually the part of code that is always executed towards the end of the program. Once the fault is injected the simulator monitors the program to check if it reaches the feedback line or gives an error before reaching it.

Feedback line can be updated by selecting a line from the source code text box and clicking the ”Update Feedback” button. The simulator informs the user about the updated feedback line through text interface as seen in Figure 10.

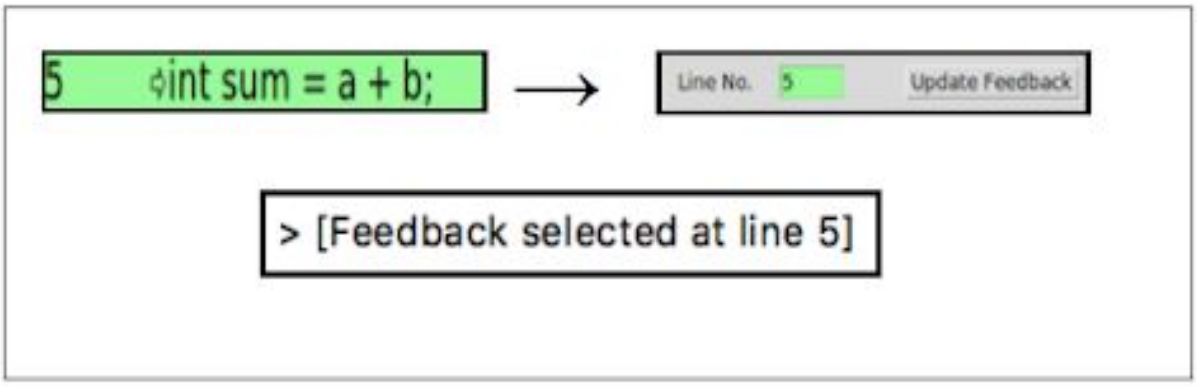

Figure 10. Feedback line update and associated output.

The simulator logs all activity completed within the application and stores the data on a local drive as text files. This allows the users to analyse the test results in detail and keep a record for each test carried out. Additionally, the logs are designed to assist users with tracing the fail point for each fault. An example of a log with a mitigated fault and a failure can be seen in Figure 11.

```
                                    ***** LOGFILE *****

                                #####  Import and Connect  #####

> [ Connected < source_file.c > Successfully ...  ]
> [ Connected to GDB Server ]
> [ Connected < xml_file.xml > Successfully ...  ]

                                 #####  Select Feedback  #####

> [ Feedback selected at line 25 ]
> [ Registers Refreshed ]
> [ Executing Breakpoints ]

                                 #####  Successful Fault  #####

> [ Executing Fault 1 from XML at 0x00010614 ]
> [ B *0x00010614 ]
> [ Breakpoint 1 at 0x10614: file /source_file.c, line 13. ]> [ continue ]
> [ Continuing. ]
> [ Breakpoint 1, main () at /source_file.c:13 ]
> [ 13      if(a < 0) decrease = false; ]
> [ Delete all breakpoints? (y or n) [answered Y; input not from terminal] ]
> [ B 25 ]
> [ Breakpoint 2 at 0x106b4: file /source_file.c, line 25. ]
> [ set $r0=5 ]
> [ set $r4=-61520690 ]
> [ set $r3=-5 ]
> [ Continuing. ]
> [ Breakpoint 2, main () at /source_file.c:25 ]
> [ 25     printf("Feedback Line\n"); ]
> [ SUCCESS Reached Feedback Line ]

                                   #####  Failed Fault  #####

> [ Executing Fault 2 from XML at 0x0001064c ]
> [ B *0x0001064c ]
> [ Breakpoint 1 at 0x1064c: file /source_file.c, line 16. ]
> [ Continuing. ]
> [ Breakpoint 1, 0x0001064c in main () at /source_file.c:16 ]
> [ 16     {  a = a - 1; ]
> [ Delete all breakpoints? (y or n) [answered Y; input not from terminal] ]
> [ B 25 ]
> [ Breakpoint 2 at 0x106b4: file /source_file.c, line 25. ]
> [ set $r2=102 ]
> [ set $r10=618448 ]
> [ set $r11=-61520646 ]
> [ Continuing. ]
> [ Program received signal SIGSEGV, Segmentation fault. ]
> [ 0x00010650 in main () at /source_file.c:16 ]
> [ 16     { a = a - 1; ]
> [ FAILED to reach FeedBack ]

                                   ***** END OF LOGFILE *****
```

Figure 11. Logging Successful and Failed Faults

5.1.5 Application Graphical User Interface

The graphical user interface for the simulator was created using the Python Tkinter library. We utilized frames, titles and dialog boxes to build this application. We used the grayscale system theme for our application and utilized minimal color to provide feedback to the user and show application status. In this subsection we discuss different components of GUI and its uses. Figure 12 displays the GUI of our final product.

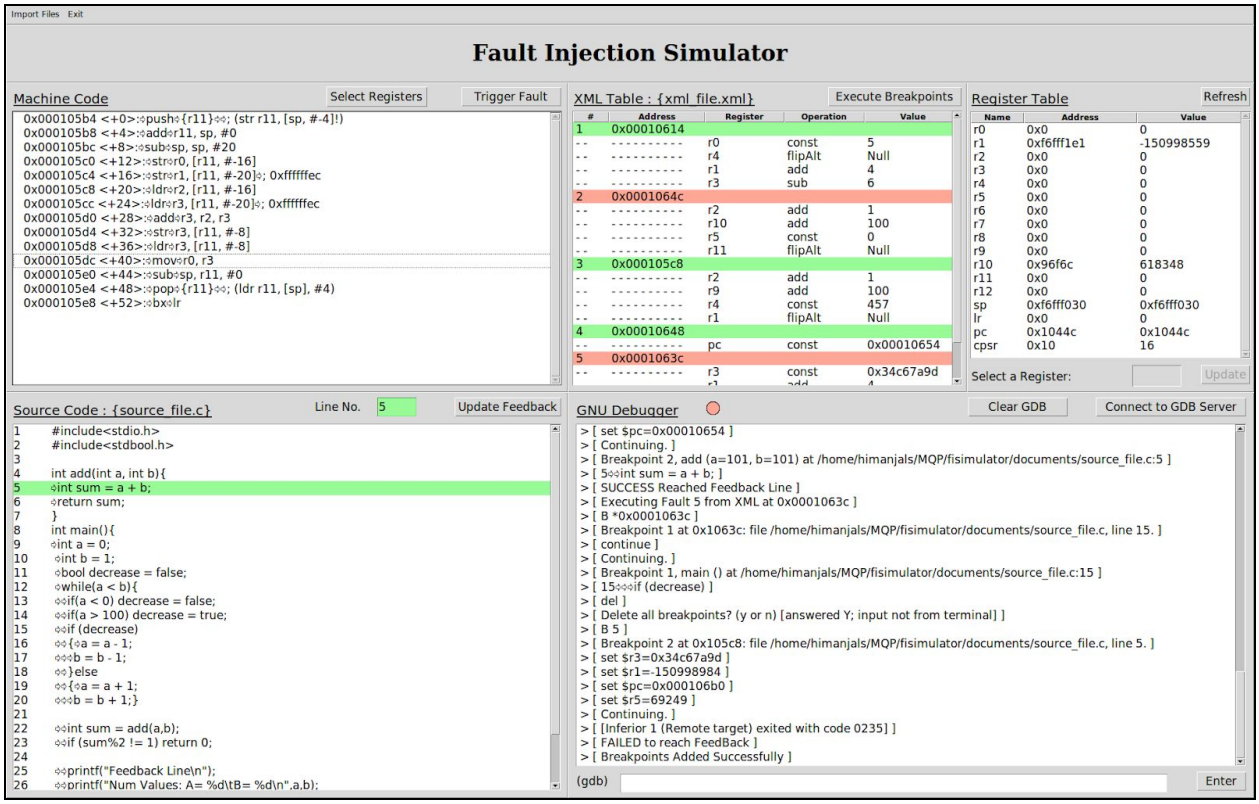

Figure 12. Fault Injection Simulator Application

5.1.5.1 Importing Source Code

A user can import source files by clicking “Import Source File” button on the top left menu button. All the text within the source file is displayed line by line in the source code listbox. A user can then select any line in the source code and a disassembled machine code for its scope will be available in the machine code listbox (see Fig. 13). Once the machine code is displayed, the user can select one or multiple instructions as trigger points.

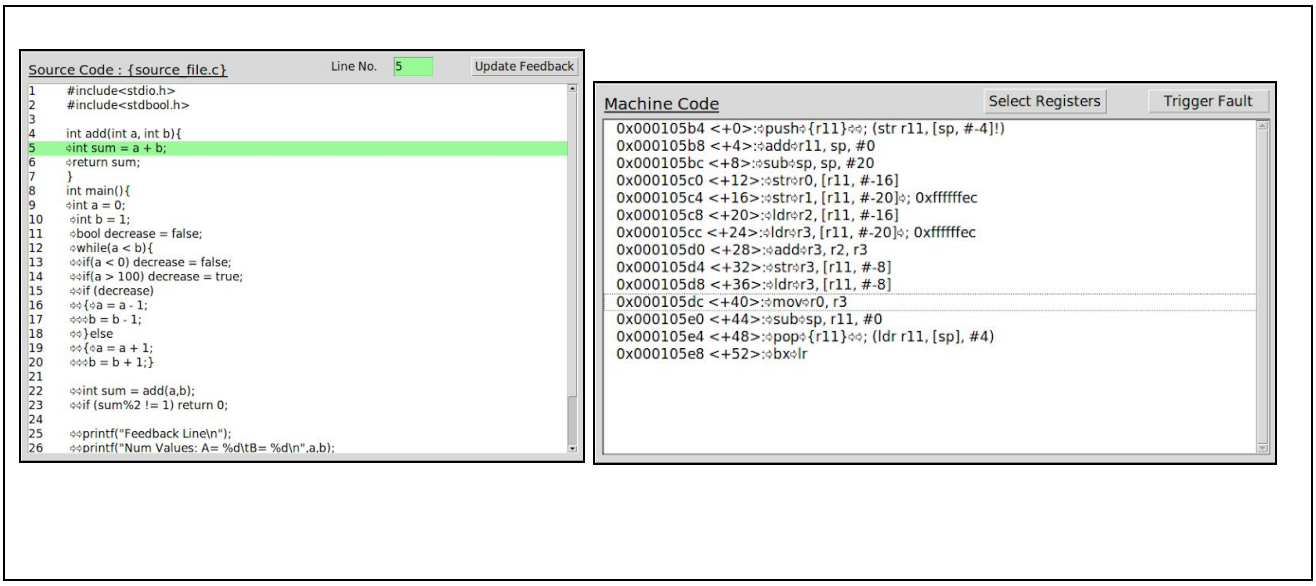

Figure 13. Source and Machine Code.

A user can also select any line in the source code and click the “Update Feedback” button to set that line as feedback line for the simulation. Users are prompted to select the registers to edit during fault simulation from the drop down menu on top of the machine code listbox. The users can select from a list containing thirteen core registers and program registers as shown in Figure 14.

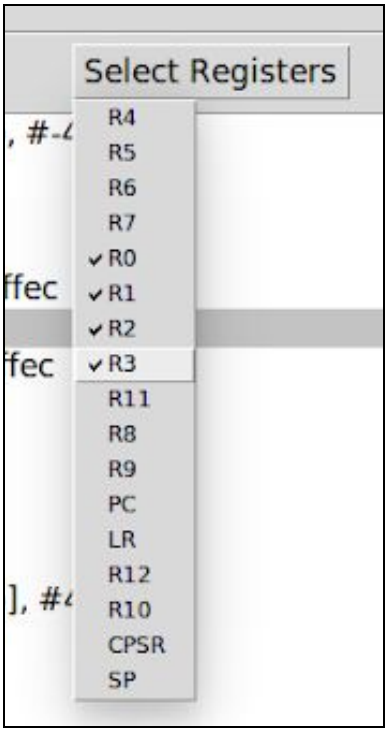

Figure 14. Register drop down.

5.1.5.2 XML Table

We used XML documents to store the glitch traces for execution within the application. We used XML as opposed to other file types due to its simplicity in reading and storing data with multi-relational properties. These XML files are generated by taking the results of a hardware fault injection trace and then used by our tool to replicate the glitch in software. The trace records breakpoint addresses and the respective registers to be edited at each breakpoint. It also keeps track of the operation and value to apply to each register. An example of a single trigger is shown in the Figure 15 below.

```
<xml>
    <fault>
        <addr breakpointAddress="0x00000000"/>
        <trigger>
            <mask> <rg register="r0"/> <mk op="const" val="0xEB80746F"/> </mask>
            <mask> <rg register="r1"/> <mk op="const" val="0x0080000B"/> </mask>
            <mask> <rg register="r2"/> <mk op="add" val="1"/> </mask>
            <mask> <rg register="r3"/> <mk op="add" val="2"/> </mask>
        </trigger>
    </fault>
</xml>
```

Figure 15. XML Document containing breakpoint address, its registers, operators and values.

Each breakpoint address has a trigger which contains the list of all the masks to apply at that breakpoint. A mask contains a register, the operation to apply to the register and the value to be used for the operation, if required. All the XML data is parsed and displayed with proper formatting (see Fig. 16).

XML Table : {xml_file.xml}

Execute Breakpoints

#	Address	Register	Operation	Value
1	0x00010614			
--	----------	r0	const	5
--	----------	r4	flipAlt	Null
--	----------	r1	add	4
--	----------	r3	sub	6
2	0x0001064c			
--	----------	r2	add	1
--	----------	r10	add	100
--	----------	r5	const	0
--	----------	r11	flipAlt	Null
3	0x000105c8			
--	----------	r2	add	1
--	----------	r9	add	100
--	----------	r4	const	457
--	----------	r1	flipAlt	Null
4	0x00010648			
--	----------	pc	const	0x00010654
5	0x0001063c			
--	----------	r3	const	0x34c67a9d

Figure 16. XML Table containing information from a sample XML document.

We used a treeview for XML data due to its easy integration with grid view. We use the same interface to provide feedback for all breakpoints by changing their color from white to red or green depending on failure or success.

5.1.5.3 Register Table

Registers are underlying memory spaces where all the variables of program are stored. Every program uses seventeen core registers namely `r0 – r12`, `Stack Pointer`, `Link Register`, `Point Counter and Current Program Status Register`. We display them using a TreeView, similar to XML Table, to divide registers in their respective address and value columns. We used these values that can be edited to simulate a fault in source code. The user can keep track of changing registers as the program runs to see changes in real time. They can also change values of registers from the textbox below the register pane. Figure 17 shows the register table as it appears to the user.

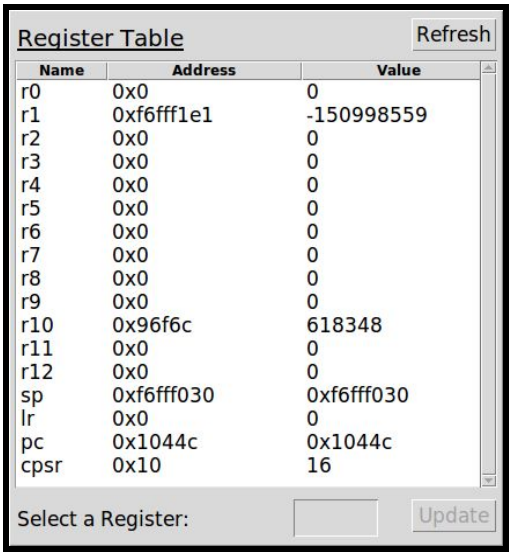

Figure 17. Register Table

5.1.5.4 GNU Debugger

While simulating faults the fault simulator sends and receives important data. Data is transferred when importing files, executing breakpoints, updating registers, stepping into instructions, disassembling code and more. We used a listbox to display all the commands sent and data received from the debugger. All debug messages and warnings are also displayed on the GNU debugger listbox (see Fig. 18).

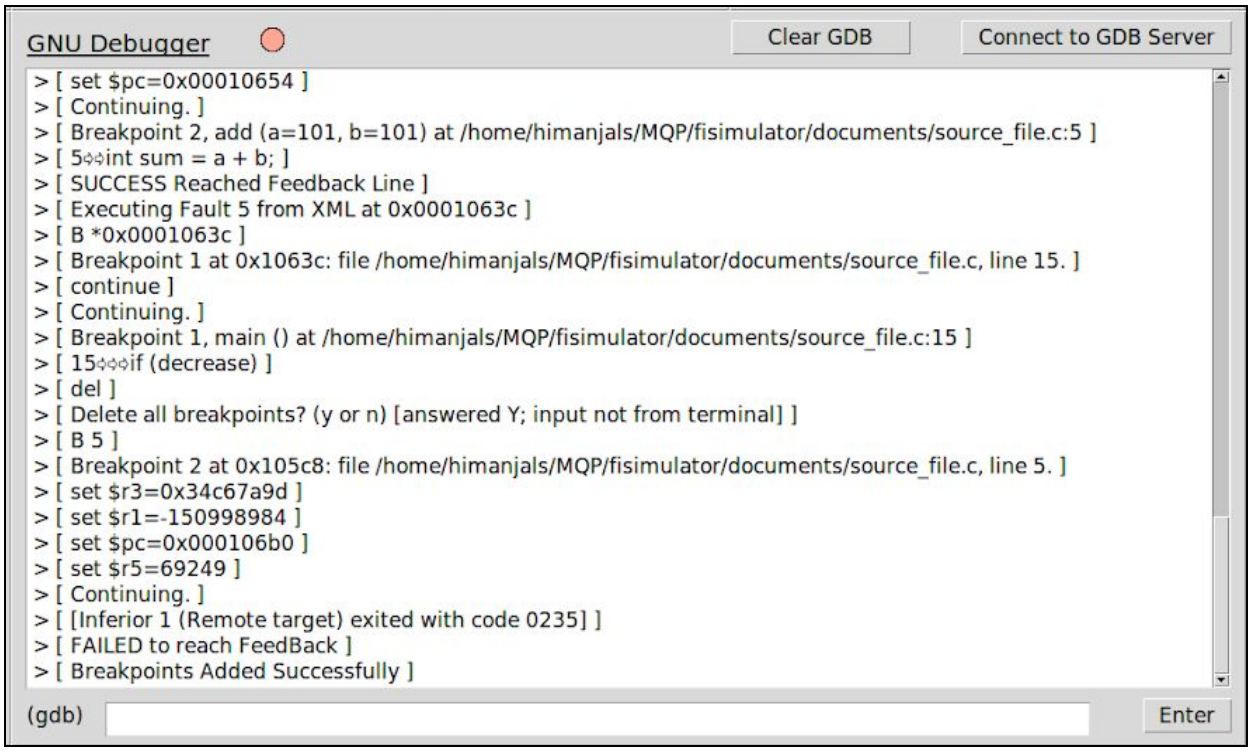

Figure 18. GNU Debugger

We also allow users to manually send commands to the GDB using the command line interface (CLI) provided below the debugger listbox. All these commands and their output are sent to the debugger console. We provide the user

with a light indicator displaying the state of the program. All states of the indicator are displayed in Figure 19. Additional functionality available through the CLI includes clearing the displayed output and reconnecting to the GDB server.

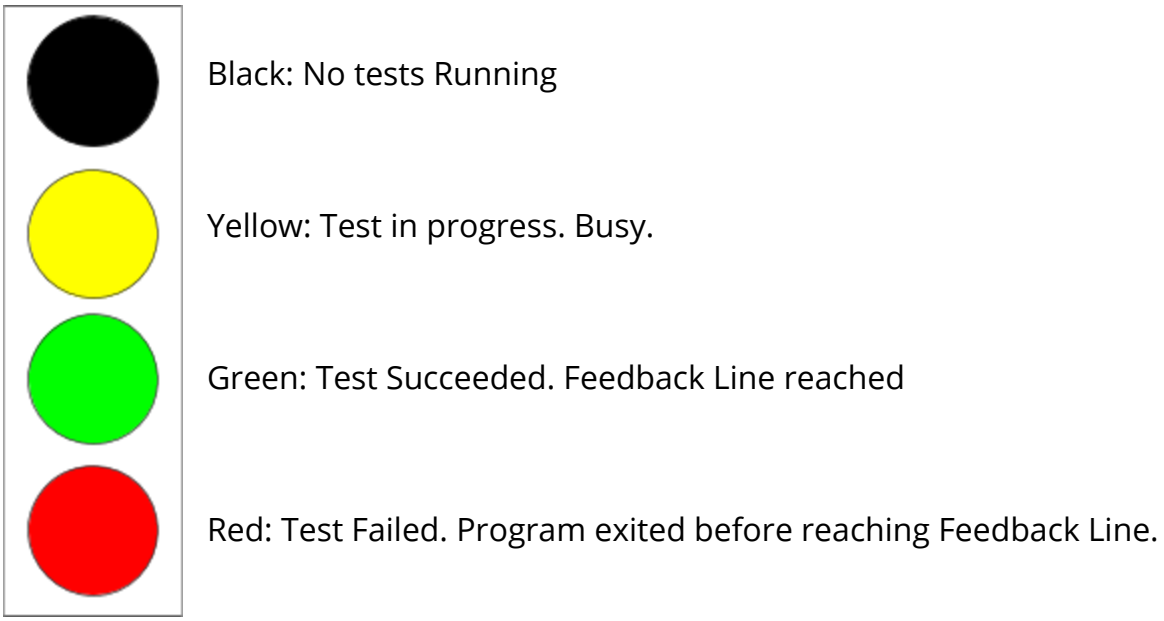

Figure 19. Initial, running, success, and failure state indicators.

5.1.6 Assessment

We developed a prototype of the application and tested it against basic C programs. We wrote a C program to swap two variables indefinitely and exit the program if the variable values are not as expected. During testing our application injected faults and deceived the program into skipping instructions and exiting the execution abnormally. We managed to do so by manipulating the program counter through the trigger fault feature of the application and skipping the swap instruction. This led to failure of variable swap and the program ended unexpectedly.

In order to test the application for multiple faults at the same time we created an XML file and added data for various faults to be injected into the C program. We loaded the swap program and the XML file into the simulator and triggered all faults using the execute XML feature of the application.

While testing the application we came across few bugs and fixed them:

1. Monitor execution at each step

   While triggering faults on multiple instructions the simulator edits registers at each instruction and steps to the next instruction. In a few cases the source program exits before the simulator can finish triggering all the instructions causing the simulator to crash. This bug was later fixed by monitoring the program execution after editing registers at each step.

2. Validate fault trigger point

   Each trigger point must exist within the scope of a program otherwise the simulator will never inject a fault and will forever wait for the execution to reach the trigger point. This bug was later fixed by validating the trigger point to be within the program memory address scope. If the point is out of scope then the fault is not simulated.

5.2 Application Implementation

After completing our initial design, we began implementing our application. We automated the simulator to allow users to test their source code for fault vulnerabilities by triggering faults. Faults were successfully simulated by changing register values and observing how the code reacts to glitches in real time. Our application can also be used to parse multiple fault traces and automatically recreate fault behavior while the source code executes. This section describes how we utilized the software and development tools we selected and the front-end and back-end development processes.

5.2.1 Software and Development Tools

To create the functionality of our application, we used various software and development tools. These include the Ubuntu Linux operating system and Python. We utilized GDB for ARM to communicate with the program file and to inject faults.

5.2.1.1 Linux and Python

We used Ubuntu throughout the development process. The simulator was developed in Python due to its simplicity in executing linux shell subprocesses. We used Python version 2.7 with Tkinter library for the GUI. We used Python 2.7 over version 3.4 due to increased documentation about Qemu and GDB support.

5.2.1.2 GDB and ARM

Source code is first compiled using *arm-linux-gnueabi-gcc*, ARM gcc for linux. The simulator then runs with the compiled arm executable in order to use the debugger. We used *arm-none-eabi-gdb* debugger from the ARM toolchain for linux to connect to Qemu which provides a remote GDB server. The ARM debugger uses a command line to interact with the server which sends and receives information for the application. A user can clear the display and reconnect to the server at any time. Table 7 shows some of the available commands.

Table 7
Sample GDB Commands.

Breakpoints	**breakpoint < address >**	add a breakpoint at `address`
	info breakpoints	view all breakpoints
	continue	continue to next breakpoint
Registers	**info Register < register >**	Refreshes `register`(s)
	set $ < reg > = < val >	Set `register` to a `value`
Machine	**disassemble < address >**	View assembly code at `address`
	si	Step into instruction
Source	**info locals**	View all variables in scope
	step	Step to next line

5.2.2 Frontend Process

We used Tkinter Python libraries to develop the interface for the fault simulator. The Tkinter features we used include frames, titles, buttons, listboxes, treeviews, entries and colors. Frames are divided into menu, title, machine code, source code, XML table, registers and GNU debugger. The menu is used for importing documents and exiting the application. The title includes the name of our application.

5.2.2.1 Listboxes

Machine code, source code and the GNU debugger simulator components were built using listboxes. Listboxes are used to display content line by line, whether its code or terminal output.

5.2.2.2 Treeview

A treeview differs from listboxes because it sorts parsed information into respective columns. The XML table and registers table were built using treeviews. The XML Table displays information from the imported XML Document in tabular form. It is sorted into breakpoints, registers, operations and values. The register table displays all the registers with their updated values in columns of register, address and value. A user has the option to select a register to update its value.

5.2.2.3 Feedback Colors

We decided to show the user more information about our tool through color. We chose black, yellow, green and red, each to indicate initial, busy, success and failure state, respectively. If a user selects or inputs the feedback line number correctly, the line and entry box is highlighted in green. This shows that the user inputs a valid line number. If the user inputs an invalid line number, the entry box highlights red.

Color is also used in the feedback bulb in the GNU Debugger frame. It is initially black when a source line is clicked and yellow when processing the fault or breakpoints. The tool then tells the user if the triggered fault executed successfully or if it failed. When executing batch faults, the breakpoint address in the XML table is highlighted green or red to indicate if the fault was mitigated successfully or not.

5.2.3 Backend Process

Once the source file is selected, it is compiled with a gcc ARM compiler “arm-linux-gnueabi-gcc”. Qemu is launched as a subprocess with debugging options and single-stepping on a given port number. The GNU debugger from the ARM toolchain is initialized as a subprocess. All the instructions and data are piped to and from the command line.

5.2.3.1 Source Code

Whenever the user clicks on a source code line in the interface, the application generates the machine code for the line's scope and displays it in the machine code listbox. This allows the user to look at the breakdown of source code into assembly instructions and to select the trigger point with more accuracy. In order to generate the machine code for the selected line scope, we add a breakpoint at the source code line to get the program address at the given line. We use the debugger to generate machine code at the given address and update the listbox with new data (see Fig. 20).

Figure 20. Source to Machine Code

We required some process in the source code to check if the code is migating faults successfully. We allow the user to select a line in the source code to denote it as the feedback line (see Fig. 21). A feedback line is ideally a part of source code that will always be executed unless the program is behaving abnormally. We add a breakpoint at the feedback line and run the program. Once the trigger breakpoint is

reached and the fault behavior is introduced, if the program succeeds to reach the feedback breakpoint without a run time error it is considered to be running successfully.

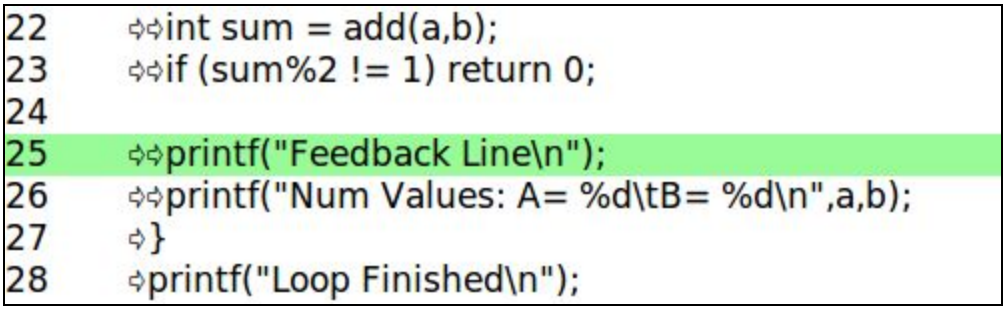

Figure 21. Select Valid Feedback Line

5.2.3.2 Registers

We added a registers panel to the interface to show the users information about changing register values. In order to get the values of all the registers we send a `info Register` command to the gdb server. To get the value for a specific register we send `info Register <register>`. We also allow users to set register values from the interface. To update a register value we send command `set $<register> = <new value>`.

A fault is simulated by changing specific register values. For example, one can change the value of program counter register and cause the program to skip some important instructions. We can also edit the values of the core registers to change the value stored in local variables of the program.

5.2.3.3 Mask

We applied a mask to registers in order to edit their values and thereby simulate a fault. A mask is a pattern which is applied to a register value to generate a new value. We developed multiple masks, each of which manipulate registers on an integer, hexadecimal or bit level (see Table 8).

Table 8
Supported mask operations.

`add`	adds an integer value to register value.
`sub`	subtracts and integer value from register value.
`const`	sets the value of register to the value of constant (integer or hexadecimal).
`FlipAlt`	Flips the alternate bits of the register binary value.

5.2.4 Trigger a Fault

A fault is simulated in our application by changing the values of underlying registers and introducing errors in the flow of a program. We allow the user to select a feedback line, trigger point and the registers for fault simulation. Once the user selects all the options and triggers a fault we start the automated fault trigger process.

A fault is simulated in five steps, as depicted in Figure 22:

1. Connect to Qemu
2. Reach the Trigger Point
3. Change Registers
4. Trigger next instruction
5. Get Feedback

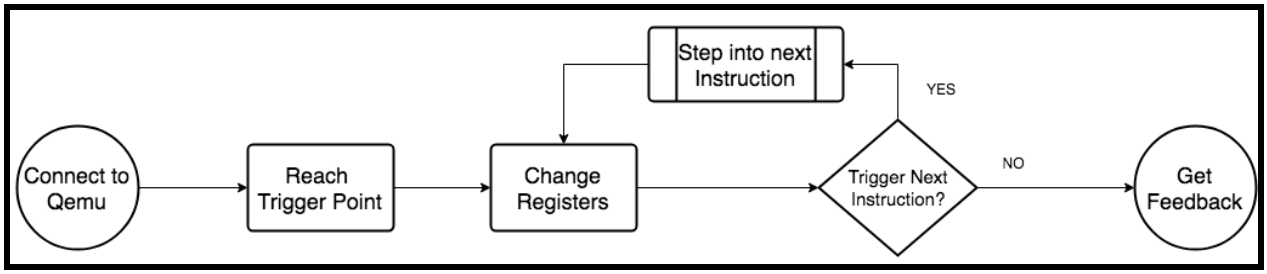

Figure 22. Process of Triggering a Fault

5.2.4.1 Connect to Qemu

The application launches Qemu on a specific port with debugging options and single stepping. Once launched, the debugger client is attached to a gdb server running inside Qemu.

5.2.4.2 Reach the trigger point

Before executing the program, breakpoints must be added at trigger points and the user-selected feedback line in order to pause execution at those addresses. Once both breakpoints are added, program execution resumes and continues until the program counter reaches the first trigger point.

5.2.4.2 Change Registers

Once the program reaches the first trigger point, program execution is halted to modify register values. The users selects the registers to edit before triggering the fault. The value at each selected register is then extracted and a mask is applied to the value. The selected register are updated with the new value by sending the command `set $<register> = <new value>`.

5.2.4.3 Trigger Next Instruction

If the user selected more than one instruction to trigger, the application stepped through every instruction and changed the registers as required. This step is repeated until there are no more instructions to trigger.

5.2.4.4 Get Feedback

Once all the selected lines have been reached and all the registers changed, program execution resumes until the feedback line is reached. If the program reached the feedback breakpoint successfully, the glitch is considered mitigated and feedback is provided for a successful test. Otherwise, if the feedback breakpoint was not reached because the program execution ended early due to an error, such as a segmentation fault or run-time error, the application reports the test as a failure.

5.2.4.4 Executing XML

A user can execute multiple trigger points on a source file at once. In order to do this the trigger data is imported from a XML file and executed. Once the test starts the simulator reads one fault at a time from the data and simulates a glitch at the trigger point. The registers requiring updates during simulation are read from the XML file, as opposed to the the drop down menu which is used while triggering a single fault. The simulator repeats the process for each fault in the XML file, restarting the virtualizer and the program after every fault. As the faults are simulated and their feedback is received, the simulator updates the user by coloring the XML line green (success) or red (failure).

6. Conclusions and Future Work

Fault injection attacks pose a vulnerability to software integrity. Testing for fault injection vulnerabilities is a common practice when creating systems with a hardware component. Hardware fault injection is expensive and difficult to scale whereas a software simulation approach adds flexibility, protects hardware and simplifies testing procedures.

There are consequences and challenges associated with voltage fault injection attacks and fault injection testing. Significant time is spent determining if hardware has been damaged when testing using voltage fault injection. Hardware testing is difficult to scale because it requires substantial resources. Unlike hardware fault injection testing, simulated fault injection does not require hardware to be replaced or available.

Our project aimed to understand the impact of voltage fault injection on hardware and create a data driven fault injection simulator to assist with testing. We succeeded in making a working fault injection simulator prototype which could send instructions into a virtualized environment to inject faults into the simulation. The simulator changes register values in a defined pattern to replicate glitches. The simulator we built supports addition, subtraction, constant, and bit-flipping operations to be applied to registers. We presented key findings from our hardware study and our simulator to the Nvidia team. We created comprehensive documentation to accompany our simulator and completed a code review prior to

committing our final product. We concluded our project by submitting our generic fault injection simulator.

Our hardware study observed the behavior resulting from voltage fault injection on a small sample of instructions and values. We observed negative values may be more vulnerable to voltage fault injection, that load instructions are sensitive to voltage fault injection, and that branching the current instruction may be protected.

It would be beneficial to conduct a similar study on a larger scale with many more test codes. Additionally we recommend that future engineers focus on the following:

1. Automate glitch detection, pattern detection and trace generation.
2. Automate glitch parameter selection.

We detected glitches using conditional formatting in Google Sheets. Building an application to complete this task may be worthwhile for future testing. Patterns less easily detected by the human eye could be identified and monitored over time by automating this process. The fault injection simulator would be most useful if traces were automatically generated based on real hardware study results. One of the most time consuming challenges during the hardware study was determining the appropriate glitch parameters. This process could be automated with a script to launch CW Capture with various parameters, with the prerequisite that glitch detection is automated.

Another potential area of future focus is additional masks and operators for more accurate fault injection simulations. Due to limited time, we used Qemu exclusively throughout the project. The application must be edited and ported to Synopsys VDK instead of Qemu virtualizer. This would enable Nvidia engineers to test larger programs like SOCs (System on Computer) that are incompatible with Qemu.

References

Ltd., Arm. “DSTREAM – Arm Developer.” ARM Developer, Arm Holdings, Retrieved February 27, 2017. URL: http://developer.arm.com/products/software-development-tools/debug-probes-and-adapters/dstream.

A. Barenghi, G. Bertoni, E. Parrinello and G. Pelosi, "Low Voltage Fault Attacks on the RSA Cryptosystem," *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Lausanne, 2009, pp. 23-31. Retrieved February 15, 2017. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5412860&isnumber=5412840

C. Burch, “ARM Subroutines & Program Stack.” Oct. 2012. Retrieved February 27, 2018. URL: http://www.toves.org/books/armsub/#s1

J. V. Carreira, D. Costa and J. G. Silva, "Fault Injection Spot-Checks Computer System Dependability," in IEEE Spectrum, vol. 36, no. 8, pp. 50-55, Aug 1999. Retrieved December 12, 2017. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=780999&isnumber=16946

GNU, "What is GDB?" November 1, 2017. Retrieved December 11, 2017. URL: https://www.gnu.org/software/gdb/

"Hackers Claim to Have Beaten Xbox 360 Security." *Hackers Claim to Have Beaten Xbox 360 Security - The H Security: News and Features*, Aug. 30 2011. Retrieved February 28, 2017. URL: www.h-online.com/security/news/item/Hackers-claim-to-have-beaten-Xbox-360-security-1333597.html

Mei-Chen Hsueh, T. K. Tsai and R. K. Iyer, "Fault Injection Techniques and Tools," in Computer, vol. 30, no. 4, pp. 75-82, Apr 1997. Retrieved December 15, 2017. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=585157&isnumber=12687

M. Kooli and G. Di Natale, "A Survey on Simulation-Based Fault Injection Tools for Complex Systems," *9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, Santorini, 2014, pp. 1-6. Retrieved December 12, 2017. URL: http://ieeexplore.ieee.org.ezproxy.wpi.edu/stamp/stamp.jsp?tp=&arnumber=6850649&isnumber=6850634

Qemu, "About Qemu". 2018. Retrieved March 10, 2018. URL: https://www.qemu.org/

L. Rivière, Z. Najm, P. Rauzy, J. L. Danger, J. Bringer and L. Sauvage, "High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures," *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, Washington, DC, 2015, pp. 62-67. Retrieved February 14, 2018. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7140238&isnumber=7140225

Synopsys Inc. “VDK Family for ARM Processors”. Mar 2017. Retrieved December 17th 2017. URL: https://www.synopsys.com/verification/virtual-prototyping/vdk/vdk-for-arm.html

N. Timmers, A. Spruyt and M. Witteman, "Controlling PC on ARM Using Fault Injection," *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Santa Barbara, CA, 2016, pp. 25-35. Retrieved December 13, 2017. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7774479&isnumber=7774465

YouTube. "Reset Glitch Hack on Slim By GliGli". Online video clip. *YouTube*, 24 August 2011. February 14, 2018.

Appendix A. Hardware Study Test Codes and Results

Test 1 & 2: Copy Small Constants and Loop

```
...
asm volatile(
      "loop:"
      "mov r0, #10 \n\t"
      "mov r1, #11 \n\t"
      "mov r2, #12 \n\t"
      "mov r3, #13 \n\t"
      "mov r4, #14 \n\t"
     "b loop"
);
...
```

Listing A.1. Test 1: Copy Small Positive Constants Loop Inline Assembly Code

```
...
asm volatile(
      "loop:"
      "mov r0, #-10 \n\t"
      "mov r1, #-11 \n\t"
      "mov r2, #-12 \n\t"
      "mov r3, #-13 \n\t"
      "mov r4, #-14 \n\t"
      "b loop"
);
...
```

Listing A.2. Test 2: Copy Small Negative Constants Loop Inline Assembly Code

Table A.1
Test 1 & 2: Overview

	Test 1 (Positive Constants)		**Test 2 (Negative Constants)**	
Normal Execution	744	74.4%	626	62.6%
Glitch Detected	247	24.7%	349	34.9%
Unknown	9	0.9%	25	2.5%

Table A.2
Test 1: Enumerated Register Values in Normal Range

R0	**R1**	**R2**	**R3**	**R4**
0x0000000A	0x0000000B	0x0000000C	0x0000000D	0x0000000E

Table A.3
Test 2: Enumerated Register Values in Normal Range

R0	R1	R2	R3	R4
0xFFFFFFF6	0xFFFFFF5	0xFFFFFFF4	0xFFFFFFF3	0xFFFFFFF2

Table A.4
Test 1: Enumerated Register Values Indicating Glitch Occurred

R0	R1	R2	R3	R4
0xEB80746F	0x0080000B	0x0000000D		0x0000000F
0xEB81746F	0x0080000F	0xFEEAFFFE		
0xEBA0746F		0xFEEAFFFF		
0xEBA1746F				

Table A.5
Test 2: Enumerated Register Values Indicating Glitch Occurred

R0	R1	R2	R3	R4
0x0000000A	0x40007A5C	0x00000000	0x079F698B	
0x00200001	0xFFFEFFF1	0x000000F8	0x40035B59	
0x1FFFFFF2	0xFFFEFFF5	0x000000FF	0xFFFFFFF0	
0x7000E400	0xFFFFFFF4	0x00008000	0xFFFFFFF2	
0x70016400		0x00008400	0xFFFFFFF4	
0xEB81746B		0x40009D84		
0xEBA1746B		0x70016000		
0xEBC1746B				
0xEBE1746B				
0xFFFFFFB0				
0xFFFFFFF0				
0xFFFFFFF4				

Table A.6
Test 1: Enumerated Program Counter Values and Corresponding Disassembly

NORMAL		GLITCHED	
PC	**Disassembly**	**PC**	**Disassembly**
0x40010168	MOV r0,#0xa	0x3E81C1B0	B {pc} ; 0x3e81c1b0
0x4001016C	MOV r1,#0xb	0x3E81C230	B {pc} ; 0x3e81c230
0x40010170	MOV r2,#0xc	0x4001085C	B {pc} ; 0x4001085c
0x40010174	MOV r3,#0xd	0x40210AA8	B {pc} ; 0x40210aa8
0x40010178	MOV r4,#0xe	0x41010AA8	B {pc} ; 0x41010aa8
0x4001017C	B {pc}-0x14	0x41022EA8	B {pc} ; 0x41022ea8

Table A.7
Test 2: Enumerated Program Counter Values and Corresponding Disassembly

NORMAL		GLITCHED	
PC	**Disassembly**	**PC**	**Disassembly**
0x40010168	MVN r0,#0x9	0x00008008	B {pc} ; 0x8000
0x4001016C	MVN r1,#0xa	0x00100080	SUBS sp,sp,#1
0x40010170	MVN r2,#0xb	0x3E010AA8	B {pc} ; 0x3e010aa8
0x40010174	MVN r3,#0xc	0x3F810AAC	B {pc} ; 0x3f810aac
0x40010178	MVN r4,#0xd	0x3F81C1AC	B {pc} ; 0x3f81c1ac
0x4001017C	B {pc}-0x14 ; 0x40010168	0x3F81C1B0	B {pc} ; 0x3f81c1b0
		0x3F81C1B8	B {pc} ; 0x3f81c1b8
		0x3F81C234	B {pc} ; 0x3f81c234
		0x3F81C238	B {pc} ; 0x3f81c238
		0x3F82C234	B {pc} ; 0x3f82c234
		0x3FACC4EC	B {pc} ; 0x3facc4ec
		0x4001085C	B {pc} ; 0x4001085c
		0x40210AA8	B {pc} ; 0x40210aa8
		0x41010EA8	B {pc} ; 0x41010ea8
		0x41012EA8	B {pc} ; 0x41012ea8
		0x41062F28	B {pc} ; 0x41062f28
		0xFF93FFFE	DCI 0xeaff ; ? Undefined
		0xFFB1FFFE	DCI 0xeaff; ? Undefined

Test 3 & 4: Load Large Alternating Bit Constants and Loop

```
...
asm volatile(
      "loop:"
      "ldr r0, =0x55555555 \n\t"
      "ldr r1, =0x55555555 \n\t"
      "ldr r2, =0x55555555 \n\t"
      "ldr r3, =0x55555555 \n\t"
      "ldr r4, =0x55555555 \n\t"
      "b loop"
);
...
```

Listing A.3. Test 3: Load Large Positive Constant (Odd-Bits Set) Loop Inline Assembly Code

```
...
asm volatile(
      "loop:"
      "ldr r0, =0xAAAAAAAA \n\t"
      "ldr r1, =0xAAAAAAAA \n\t"
      "ldr r2, =0xAAAAAAAA \n\t"
      "ldr r3, =0xAAAAAAAA \n\t"
      "ldr r4, =0xAAAAAAAA \n\t"
      "b loop"
);
...
```

Listing A.4. Test 4: Load Large Negative Constants (Even-Bits Set) Loop Inline Assembly Code

Table A.8
Test 3 & 4: Overview

	Test 3 (Positive, Odd-Bits Set)		**Test 4 (Negative, Even-Bits Set)**	
Normal Execution	355	35.5%	22	2.2%
Glitch Detected	619	61.9%	966	96.6%
Unknown	26	2.6%	12	1.2%

Table A.9
Test 3: Enumerated Register Values in Normal Range

R0	**R1**	**R2**	**R3**	**R4**
0x55555555	0x55555555	0x55555555	0x55555555	0x55555555

Table A.10
Test 4: Enumerated Register Values in Normal Range

R0	R1	R2	R3	R4
0xAAAAAAAA	0xAAAAAAAA	0xAAAAAAAA	0xAAAAAAAA	0xAAAAAAAA

Table A.11
Test 3: Enumerated Register Values Indicating Glitch Occurred

R0	R1	R2	R3	R4
0x00000000	0x00000000	0x00000000	0x079F698B	0x00000000
0x7000E400	0xD5555555	0x000000FF	0x5D555555	0xE59F55BC
0xD5555555	0xD5D55555	0x00008000	0xDD555555	0xE69E7521
0xEB805CFF	0xE59F3618	0xD5555555	0xDD555575	
0xEB8105F6	0xE5D33563	0xD5D13561	0xE59F2004	
0xEB815CFF	0xE5D335E3	0xD5D55555	0xE59F301C	
0xEB815DFF	0xE5DB35E3	0xD5D93561	0xE59F4614	
		0xE59F2004	0xEAFFFFF9	
		0xE59F4614	0xEAFFFFFE	
		0xE5D13563	0xEDD331E3	
		0xE5D33163		
		0xE5D331E3		
		0xE5DB31E3		
		0xEAFFFFFE		

Table A.12
Test 4: Enumerated Register Values Indicating Glitch Occurred

R0	R1	R2	R3	R4
	0xA2CA28AA	0xA6CA29AB	R3=0xAAAAAAAB	
	0xAA8AAAAA	0xAA8AAAAA		
	0xE3D00400	0xE3D00000		
	0xE3D20400	0xE3D00004		
	0xE5D20402	0xE3D00400		
	0xE5D21502	0xE3D00404		
	0xE5D21506	0xE5933000		
	(continued)	(continued)		

	R1 (continued)	R2 (continued)		
	0xE5D215DE	0xE59F4614		
	0xE5D2C5DF	0xE5D20002		
	0xE5D315E1	0xE5D20102		
	0xE5D331E3	0xE5D21102		
	0xE5D335E3	0xE5D21106		
	0xE6D331A3	0xE5D2119E		
	0xE7D331A3	0xE5D211DE		
		0xE5D311E1		
		0xE5D331E3		
		0xE5D335E7		
		0xE7D331A3		
		0xE7D331E3		

Table A.13
Test 3: Enumerated Program Counter Values and Corresponding Disassembly

NORMAL		GLITCHED	
PC	**Disassembly**	**PC**	**Disassembly**
0x40010168	LDR r0,[pc,#1572]; [0x40010794]	0x00010010	B {pc} ; 0x10010
0x4001016C	LDR r1,[pc,#1568] ; [0x40010794]	0x00040010	B {pc} ; 0x40010
0x40010170	LDR r2,[pc,#1564] ; [0x40010794]	0x00040014	B {pc} ; 0x40014
0x40010174	LDR r3,[pc,#1560] ; [0x40010794]	0x00100080	SUBS sp,sp,#1
0x40010178	LDR r4,[pc,#1556] ; [0x40010794]	0x00100084	LDREQ r0,[pc,#28]; [0x1000A8]= 0x7000E400
0x4001017C	B {pc}-0x14 ; 0x40010168	0x00100088	MOVEQ r1,#0x10
		0x0010008C	STREQ r1,[r0,#0]
		0x00100090	B {pc}-0x10 ; 0x100080
		0x00140060	B {pc} ; 0x14005c
		(continued)	(continued)

		PC	Disassembly
		0x3E010AAC	B {pc}; 0x3e010aac
		0x3E010AB0	B {pc}; 0x3e010ab0
		0x3F010EB0	B {pc}; 0x3f010eb0
		0x40010860	B {pc} ; 0x40010860
		0xFE140060	B {pc} ; 0xfe140060

Table A.14
Test 4: Enumerated Program Counter Values and Corresponding Disassembly

NORMAL		GLITCHED	
PC	**Disassembly**	**PC**	**Disassembly**
0x40010168	LDR r0,[pc,#1572] ; [0x40010794]	0x40010860	B {pc} ; 0x40010860
0x4001016C	LDR r1,[pc,#1568] ; [0x40010794]	0xAAAAAAA6	DCI 0xeaff ; ? Undefined
0x40010170	LDR r2,[pc,#1564] ; [0x40010794]	0xE09FF3F8	B {pc} ; 0xe09ff3f8
0x40010174	LDR r3,[pc,#1560] ; [0x40010794]	0xEAFFFFFE	DCI 0xeaff ; ? Undefined
0x40010178	LDR r4,[pc,#1556] ; [0x40010794]		
0x4001017C	B {pc}-0x14 ; 0x40010168		

Test 5 & 6: Copy Small Constants and Branch Current Instruction

```
...
asm volatile(
      "mov r0, #10 \n\t"
      "mov r1, #11 \n\t"
      "mov r2, #12 \n\t"
      "mov r3, #13 \n\t"
      "mov r4, #14 \n\t"
      "b . \n\t"
      "b . \n\t"
      "b ."
);
...
```

Listing A.5. Test 5: Copy Small Positive Constants and Branch Current Instruction Inline Assembly Code

```
...
asm volatile(
      "mov r0, #-10 \n\t"
      "mov r1, #-11 \n\t"
      "mov r2, #-12 \n\t"
      "mov r3, #-13 \n\t"
      "mov r4, #-14 \n\t"
      "b . \n\t"
      "b . \n\t"
      "b ."
);
...
```

Listing A.6. Test 6: Copy Small Negative Constants and Branch Current Instruction Inline Assembly Code

Table A.15
Test 5 & 6: Overview

	Test 5 (Positive Constants)		**Test 6 (Negative Constants)**	
Normal Execution	986	98.6%	953	95.3%
Glitch Detected	0	0%	36	3.6%
Unknown	14	1.4%	11	1.1%

See Table A.2 and A.3 for the enumerated register values in normal range for test 5 and test 6, respectively. Omitted table showing the enumerating register values indicating a glitch occurred for both tests because all values for R0 through R4 were within normal range.

Table A.16
Test 5: Enumerated Program Counter Values and Corresponding Disassembly

NORMAL		GLITCHED	
PC	**Disassembly**	**PC**	**Disassembly**
0x40010168	MOV r0,#0xa		
0x4001016C	MOV r1,#0xb		
0x40010170	MOV r2,#0xc		
0x40010174	MOV r3,#0xd		
0x40010178	MOV r4,#0xe		
0x4001017C	B {pc}-0x14 ; 0x40010168		

Table A.17
Test 6: Enumerated Program Counter Values and Corresponding Disassembly

NORMAL		GLITCHED	
PC	**Disassembly**	**PC**	**Disassembly**
0x40010168	MVN r0,#0xa	0x40010180	LDR r5,[pc,#1468] ;[0x40010744]
0x4001016C	MVN r1,#0xb		
0x40010170	MVN r2,#0xc		
0x40010174	MVN r3,#0xd		
0x40010178	MVN r4,#0xe		
0x4001017C	B {pc}-0x14 ; 0x40010168		

Test 7: Copy Zero Sequence and Increment by One Loop

```
...
asm volatile("mov r0, #0 \n\t"
             "mov r1, #0 \n\t"
             "mov r2, #0 \n\t"
             "mov r3, #0 \n\t"
             "mov r4, #0 \n\t"
             "loop: \n\t"
             "add r0, r0, #1 \n\t"
             "add r1, r1, #1 \n\t"
             "add r2, r2, #1 \n\t"
             "add r3, r3, #1 \n\t"
             "add r4, r4, #1 \n\t"
             "b loop"
);
...
```

Listing A.7. Test 7: Copy Zero Sequence and Add One Loop Inline Assembly Code

Table A.18
Test 7: Overview

Normal Execution	595	59.5%
Glitch Detected	282	28.2%
Unknown	123	12.3%

Omitted tables showing the enumerated register values for test 7 because test resulted in hundreds of normal and glitched values.

Table A.19
Test 7: Enumerated Program Counter Values and Corresponding Disassembly

NORMAL		GLITCHED	
PC	**Disassembly**	**PC**	**Disassembly**
0x4001017C	ADD r0,r0,#1	0x00100084	LDREQ r0,[pc,#28] ; [0x1000A8] = 0x7000E400
0x40010180	ADD r1,r1,#1	0x00100090	B {pc}-0x10 ; 0x100080
0x40010184	ADD r2,r2,#1	0x3E010188	B {pc} ; 0x3e010188
0x40010188	ADD r3,r3,#1	0x3E014198	B {pc} ; 0x3e014198
0x4001018C	ADD r4,r4,#1	0x3E01C198	B {pc} ; 0x3e01c198
0x40010190	B {pc}-0x14 ; 0x4001017c	0x3E01C598	B {pc} ; 0x3e01c598
		0x3E02C5E0	B {pc} ; 0x3e02c5e0
		0x3E05418C	B {pc} ; 0x3e05418c
		0x3E25418C	B {pc} ; 0x3e25418c
		0x40010870	B {pc} ; 0x40010870

Test 8: Alternate Loading Small Constants Loop

```
...
asm volatile("loop: \n\t"
             "ldr r0, =0x00000000 \n\t"
             "ldr r1, =0x00000000 \n\t"
             "ldr r2, =0x00000000 \n\t"
             "ldr r3, =0x00000000 \n\t"
             "ldr r4, =0x00000000 \n\t"
             "ldr r0, =0x0000000A \n\t"
             "ldr r1, =0x0000000A \n\t"
             "ldr r2, =0x0000000A \n\t"
             "ldr r3, =0x0000000A \n\t"
             "ldr r4, =0x0000000A \n\t"
             "b loop"
);
...
```

Listing A.8. Test 8: Alternate Loading Small Constants Loop Inline Assembly Code

Table A.20
Test 5: Overview

Normal Execution	742	74.2%
Glitch Detected	244	24.4%
Unknown	14	1.4%

Table A.21
Test 8: Enumerated Register Values in Normal Range

R0	R1	R2	R3	R4
0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x0000000A	0x0000000A	0x0000000A	0x0000000A	0x0000000A

Table A.22
Test 8: Enumerated Register Values Indicating Glitch Occurred

R0	R1	R2	R3	R4
0x00000008	0x00000004	0x00008400	0x40035B6D	0x40010172
0x40010172	0x00800400	0x40009D84		
0xEB814E68	0x40007A5C	0xE59FF3F8		
0xEBA14E68	0x40009DB4	0xFFFEEAFE		
0xEBA16E68		0xFFFFFEEA		

Table A.23
Test 6: Enumerated Program Counter Values and Corresponding Disassembly

NORMAL		GLITCHED	
PC	**Disassembly**	**PC**	**Disassembly**
0x40010168	MOV r0,#0	0x3E810ABC	B {pc} ; 0x3e810abc
0x4001016C	MOV r1,#0	0x3E81C21C	B {pc} ; 0x3e81c21c
0x40010170	MOV r2,#0	0x3E82C21C	B {pc} ; 0x3e82c21c
0x40010174	MOV r3,#0	0x3E82C220	B {pc} ; 0x3e82c220
0x40010178	MOV r4,#0	0x3F810ABC	B {pc} ; 0x3f810abc
0x4001017C	MOV r0,#0xa	0x40010870	B {pc} ; 0x40010870
0x40010180	MOV r1,#0xa	0x40210ABC	B {pc} ; 0x40210abc
0x40010184	MOV r2,#0xa	0x4022C8F0	B {pc} ; 0x4022c8f0
0x40010188	MOV r3,#0xa		
0x4001018C	MOV r4,#0xa		
0x40010190	B {pc}-0x28 ; 0x40010168		