# SoK: Getting started with open-source fault simulation tools

Asmita Adhikary[0000−0002−2757−1271] and Ileana Buhan[0000−0001−5494−9164]

Radboud University, Houtlaan 4, 6525 XZ Nijmegen, The Netherlands
{asmita.adhikary,ileana.buhan}@ru.nl

**Abstract.** Fault injection attacks have caused implementations to behave unexpectedly, leading to the extraction of cryptographic keys and the spectacular bypass of security features. Understandably, developers want to ensure the robustness of the software against faults and eliminate during production weaknesses that could lead to exploitation. Several open-source fault simulation tools have recently been released to the public, promising cost-effective fault evaluations. In this paper, we set out to discover how suitable such tools are for a developer who wishes to create robust software. The four fault simulation tools available to us employ different techniques to navigate faults and present varying difficulty levels to the user. We objectively compare the available open-source tools and discuss their benefits and drawbacks.

**Keywords:** Fault injection · Simulation · Open-source · Software

## 1 Introduction

An adversary with physical access to a device can induce unforeseen effects in a software program by subjecting the device to extreme operating conditions. Faults can be introduced in several ways. Examples are *clock glitches* - where short glitches are inserted into the clock signal, which may cause timing violations, *voltage glitches* - where the device is supplied with power outside the range of values specified in the datasheet, *light amplification* - by stimulated emission of radiation shots, or *temperature* variations. Modifying or skipping the intended flow of operations or tweaking data values could result in the software acting unexpectedly. Unauthorised individuals accessing off-limits memory locations or bypassing necessary authorisation conditions could crash the system or cause it to behave unexpectedly, leading to dire consequences. Fault attacks on the PlayStation hypervisor of Sony and the Xbox 360 [1] through the reset glitch provide enough incentive to prevent further attacks by injection of faults. Re-enabling debug access to EFM32WG [16] using electromagnetic fault injections or hacking Apple AirTags [2] gives us a glimpse of the consequences of fault injection attacks. Glitching the Trezor One hardware wallet [5] to extract the recovery seed leading to the wallet's cloning by exploiting faults gives further proof of the disastrous effects of fault injection attacks.

A convenient solution for developers is to run their implementations through fault simulation tools since injecting faults into the real hardware implementation is expensive, complex, and time-consuming. In addition to convenience, the benefit of using fault simulation tools is the possibility of detecting the cause that led to a successful fault. Detecting the cause of a leak is typically not possible when we perform fault injection with real hardware.

Fault simulation tools replicate the underlying architecture on which faults are supposed to be injected. This makes it possible for a user to determine the cause of a successful fault and harden the implementation without needing the device. Using fault simulators may prove less complex and expensive than testing for faults using real fault injection tooling. Given the appeal of fault simulators, we sought to investigate state-of-the-art open-source general-purpose fault simulation tools. We found several fault simulation tools available in the public domain. These tools serve different use cases. We divide fault simulators into *general-purpose* tools, which can test the resilience to faults for any software programs, and *cryptographic algorithms*, which are intended to test the resilience of cryptographic algorithms to specialized attacks. This paper focuses on general-purpose tools since their scope is broader. We set out to discover how difficult it is for a developer to get started with fault simulators, what features such tools offer, which use case scenarios they cover, and how easy it is to adapt them to different examples. Therefore, the research question we investigate in this paper is:
*Are existing open-source fault simulation tools ready to replace the hardware tool when testing general purpose software?*
**Contribution** In this paper, we objectively compare existing open-source fault simulation tools designed for general-purpose software. We define a set of parameters for comparing the tools to help a prospective user quickly decide which tools are best suited for his use case. We also propose a grammar for expressing fault models, which can be used to compare each tool's capabilities at a glance. Finally, we discuss the pros and cons of all four tools.
**Paper organization** The rest of the paper is organized as follows. Section 2 presents the development of fault injection tools over the years. Section 3 describes the criteria we chose to evaluate the four fault injection tools. Section 4 details the experimental setup, while section 5 elucidates the results as well as outlines the main features and workings of the tools, including their advantages and drawbacks. Section 6 identifies the pros and cons of using one fault injection tool over the other, followed by conclusions in Section 7.

## 2 Related Works

There are surprisingly many tools available to perform fault simulations. We first divide existing tools according to their intended use, *software* vs *hardware circuits*. We define the intended application for the verified target as the second axis: *general purpose* hardware and software vs. *cryptographic algorithms*. Lastly, we consider whether the tools are open source. As the subject of our pa-

per is general-purpose fault simulator tools for software, we only mention from this category the tools that have not been open-sourced. Due to a large number of available tools in both general purpose [3], [22], [14], [10], [17], [24] tool and cryptographic algorithm [7], [15], [21], [23], [26], [19] tool categories, in the following, we make a representative selection of tools available for general-purpose software.

**General purpose, circuits.** The first general purpose tool intended to test circuit fault resistance is MEFISTO (Multilevel Error / Fault Injection Simulation TOol) [13]. Using VHDL as the simulation language, MEFISTO validates the dependability of fault-tolerant systems by applying fault injection to different levels of abstraction, which in turn is used to create an abstraction hierarchy of fault/error models. It also estimates the possible coverage with the given fault-tolerance mechanisms. MEFISTO is not open source. LIFTING (LIRMM Fault Simulator) [6] is a tool designed in Verilog on an event-driven logic simulation engine that focuses on both logic and fault simulations for stuck-at faults and single event upsets (SEUs), and results show that its execution time is comparable to commercial tools. It takes the netlist, test sequence, and fault list to provide the test report as output. The simulator checks if the design meets the expectations of ten functional specifications and introduces fault injections for stuck-at and single-event upset fault models. LIFTING is open source. More recently, AFIASIC(DFTS) has been proposed for the automated integration of fault injection into the ASIC design flow [25].

**General purpose, software.** A language and compiler-independent framework named XEMU [4] is an extension of the QEMU software emulator and carries out efficient mutation-based testing of software binaries by injecting mutations at runtime using dynamic code translation (which prevents the software binaries from getting affected). It uses the user-mode QEMU, which emulates a single program on a Linux OS. Without access to the source code, the control flow graph (CFG) analysis of the disassembled code (before the execution of the software binary) is used to create a mutation table to facilitate the injection.XEMU approaches 100% accuracy for the test quality metrics compared to source code instrumentation. The CFG offers a speed-up of up to 100 - 1000 times with a GDB/ARMulator. An automatic, non-intrusive simulation-based fault injection framework based on QEMU is put forward in [9]. The authors note that the tool is an efficient, open-source instruction-accurate emulator for microprocessor architectures. The tool can simulate the presence of permanent, intermittent, and transient faults in the CPU registers of both RISC and CISC architectures.

QEFI (QEMU Fault Injector) [8] tool for system-wide, and kernel-based fault injection is a QEMU-based fault injection framework focused on ARM architecture. It injects faults into the operating system to check its susceptibility to faults and its reliability and robustness. Since it aims to simulate hardware faults in software environments, there remains a trade-off between manageability and scalability vs performance overhead and setup. The Tiny Code Generator (TCG) of the QEMU simulation platform (on which the tool is based) covers all possible faults. Models can be implemented for different levels of abstraction

as microarchitecture simulation, instruction-level simulation, or virtualization. However, setting up the environment for various abstraction levels can be costly, while the test results can be inaccurate. That said, since it emulates different architectures without modifying the software under test, it can be run in distributed environments. Having a python interface makes it quite user-friendly. The chronology of running experiments would start with QEMU and application run control, followed by fault injections and results processing. Being a heterogeneous environment, designing a logging mechanism is left to the lowest level possible while providing specific log management mechanisms like a log directory and a way to synchronize them with timestamps. The experiments demonstrate flaws in both the Linux kernel and QEMU itself.

## 3 Criteria for evaluating the simulators

This section discusses our criteria for evaluating the four open-source general-purpose fault simulation tools. From the user's perspective, these criteria provide a quick insight into the working of each tool to help decide which tool to use.

Table 1 is a concise overview of the four-fault simulation tools and the parameters we selected for comparison. The *type* parameter informs us of the underlying simulation or emulation engines. *Architecture* describes the supported hardware device on which the execution of the software can be replicated. The *OS* parameter instructs us about the operating system for which the fault simulation tool will run and compatible cross-compilers (together with the *architecture* parameter). *Range* specifies whether a tool is *exhaustive*(E) and will consider the entire implementation or *user-defined*(U) where the user can specify the range of instructions. *Coverage* determines whether the faults injected are *deterministic* (D), where different fault injection experiments will always produce the same results, or *random*(R), where different fault injection experiments will produce different results. The *fault models* parameter notifies us of the various possibilities of injecting faults. The problem we faced when looking for a way to compare the supported fault models was that there was no unified view. Table 2 shows our solution and describes every fault model composed of these characteristics. The first is, *how* are the effects of the fault simulation manifested: permanent, transient, or for a specified time interval. The second is *who* (or, which fragment) is affected by faults, instruction, data, or addresses and what is the *resolution*, bit, byte, or a full register. Finally, each fault model is described by an *action* with the help of which the fault is injected.

For example, Table 1 shows that FiSim supports two fault models: *transient instruction skipping* ([T][I][N]) and *transient instruction bit flips*([[T][I][b][G]]). Finally, we note that the combination in *fault models* does not imply that tools support all possible fault model combinations.

**Table 1.** Evaluation Criteria

| Tool | Type | OS | Architecture | Range | Coverage | Fault Models |
|---|---|---|---|---|---|---|
| FiSim | Unicorn | Windows (Pre-built) Linux & Mac | ARM | E | D | [T][I][b][G,N] |
| ZOFI | Native hardware | Linux | x86_64 | E | R | [T][A][b,R][G] |
| ARMORY | M-ulator | Linux | ARMv6M, ARMv7M, ARMv7EM | U | D | [*][I][*][*] |
| ARCHIE | QEMU | Linux | Depends on QEMU | U | D | [P,T][I,D][b,R] [C,S,G,F] |

**Table 2.** Fault Models with Abbreviations

| How | Who | Resolution | Action |
|---|---|---|---|
| Permanent(P)<br>Transient(T)<br>Until-overwrite(U) | Instruction(I)<br>Data(D)<br>Address(A) | Bit(b)<br>Byte(B)<br>Register(R) | Clear(C)<br>Fill(F)<br>Set(S)<br>Flip/Toggle(G)<br>Skip/NOP(instruction)(N) |

## 4 Experimental Setup

To evaluate the four fault simulation tools, we ran FiSim [1] on Windows 11 Pro, AMD Ryzen 5 PRO 5650U @ 2.30GHz and 16 GB of RAM to facilitate the use of its GUI.

We installed ZOFI (version 0.9.7) [2], ARMORY [3] and ARCHIE [4] on a VirtualBox running Ubuntu 22.04.1 LTS with one core, one thread and 10.44 GB of RAM. The VirtualBox was running on a PC with Ubuntu 20.04.4 LTS, with Intel(R) Xeon(R) CPU ES-2620 v4 @ 2.10GHz, one physical processor, eight cores, and sixteen threads with 50.43 GB of RAM. We installed the ARM GNU toolchain and Meson build system as prerequisites for running ARMORY.

To compare the tools, we first ran each tool with the default example. Next, we run the tool with a different example. Initially, we aimed to run the same code on all tools, but this was impossible due to the differences in platforms. In the next section, we report the results of our experiments.

[1] `https://github.com/Riscure/FiSim/releases`
[2] `https://github.com/vporpo/zofi`
[3] `https://github.com/emsec/arm-fault-simulator`
[4] `https://github.com/Fraunhofer-AISEC/archie`

## 5 Experimental Results

On executing the fault simulation tools with different implementations, we record their execution times. Table 3 shows the results of the experiments for each tool [5].

**Table 3.** Experimental Results

| Tool | Implementation | Binary Size | Execution Time | #Faults |
|---|---|---|---|---|
| FiSim | Secure bootloader | 5KB | 1:49.17 min | 364 |
| | Password Checker | 1KB | 40.85 sec | 139 |
| ZOFI | System file | 48KB | 55.015 sec | 74 |
| | Counter | 16KB | 26.566 sec | 82 |
| ARMORY | Fault insertion | 711B | 0.979 sec | 24 |
| | AES | 6.1KB | 24:38.730 min | 823192 |
| | Secure bootloader | 3.5KB | 24:46:39.962 hrs | 1124 |
| | Counter | 227B | 1.746 sec | 2 |
| ARCHIE | Blinking LED on STM32F0DISCOVERY | 500B | 6:16.714 min | 4 |
| | AES | 7.1KB | 2:06:58.019 hrs | 4 |
| | Blinking LED on STM32VLDISCOVERY | 148KB | 1:1.414 min | 1 |

### 5.1 FiSim

**FiSim** [20] is an open-source, deterministic fault simulator prototype based on the Unicorn emulator and the Capstone disassembler. Its GUI is pre-built for Windows but also supports Linux and Mac OSX systems. It is a proprietary closed-source tool [12] for ARM fault simulation. FiSim is made to work with the pre-loaded implementation of a secure bootloader. It is based on a cross-platform simulation of ARM32 and ARM64 architecture. FiSim implements two fault models - the transient NOP instruction model and the transient single-bit flip instruction.

The demo version comes pre-loaded with a secure bootloader implementation. The first boot stage takes the hardware model and the flash dump as input and provides the console output (if any) for debugging purposes along with the execution trace. In the second boot stage, the execution trace is used as input to provide the good and bad signatures, giving two different execution traces. Once the traces start to differ, the logic of the boot stage decides if the authentication succeeded. Then the execution trace is put in the fault generator. For every executed instruction and possible fault, the simulator is run with the fault applied to the code and the flash with the bad signature. If and when

[5] 1:49.17 min implies 1 minute and 49.17 seconds
24:46:39.962 hrs implies 24 hours, 46 minutes and 39.962 seconds

the code is vulnerable, the authentication would succeed, and the next boot stage would be executed, thus signifying that the glitch bypassed the authentication. The target source code of the secure bootloader is compiled, and the resulting binary code is used for the simulation. FiSim provides ways to harden the bootloader by pointing out its weaknesses and testing the effectiveness of its countermeasures. Adding redundancy makes the code more robust against faults. On running the default example, FiSim shows a list of assembly instructions for both fault models. Clicking on the assembly instructions takes us to the corresponding high-level instruction in the implementation. FiSim takes 1:49.17 minutes to report 364 faults.

However, loading other programs for simulation could be quite a task. The new code must follow the structure of the bootloader, or, in case of an entirely different program structure, the FiSim engine would have to be rewritten to suit the needs. The other implementation we considered for FiSim was that of a password authenticator. Similarly, as with the secure bootloader implementation, the password aunthentication procedure would verify the input password with the correct stored password. It takes 40.85 seconds to report 139 glitches.

By reusing existing components, FiSim ensures speed over accuracy and takes the shortest path to reasonable results. It iterates over all the possible faults, given the fault models, and then continues executing the target code over every possible fault. However, all potential faults aren't equally realistic or probable in the real world. So, the simulation speed gets slower with more complex fault models; it could become too slow to run the simulations in a reasonable amount of time and computational resources.

### 5.2 ZOFI

**ZOFI** (Zero Overhead Fault Injector) (version 0.9.7) [18], based on the Capstone library, is a zero-overhead open-source timing-based transient fault simulation tool. Its main feature is the speed with which it can analyze a given workload. It employs the single-event upset fault model.

At first, ZOFI executes the unmodified binary at native speed to measure its execution time and collect its original output (golden run). The execution time of the golden run serves as an upper bound for the fault injection time. It helps in approximating if the binary needs to be terminated in case it falls into an infinite loop. The outcomes of the golden run help ZOFI compare the results of the subsequent test runs to categorise the instructions as corrupted, masked, detected, stuck in an infinite loop, or throwing an exception. A corrupted instruction is one whose outputs vary from that of the golden run, whereas if an instruction doesn't terminate and takes far beyond the execution time of the golden run, which demands ZOFI to terminate it, is categorised as infinite execution. If the instruction doesn't give the same or different outcome (compared to the golden run) but throws an exception due to an illegal operation, it comes under the exception category. However, even though a fault is injected, it fails to show itself because it doesn't find use in any of the subsequent computations, in which case

ZOFI names it as masked. The previous categories will not be enough to detect exploitable faults in implementations that employ error detection mechanisms. The error detection mechanisms inform ZOFI if the injected faults cause errors and exit the binary using specific exit codes.

Then, it forks and launches a new process to run the binary, where it pauses the binary for a random period (between zero and the execution time of the golden run) to simulate faults, after which the execution of the binary is resumed. For the fault simulation, the execution of the binary gets interrupted by a signal emitted by ZOFI and it acquires access to the register states. ZOFI has access to the instruction pointer but, oblivious to the instruction type needs the Capstone library's help to determine the list of accessed registers and each access type (explicit or implicit). Then ZOFI modifies the state of the binary by injecting a register bit-flip fault. ZOFI takes a slightly different approach to introduce faults into the register depending upon whether they are read by instruction or written by an instruction - unlike in the case of the registers being read, for the registers being written to, ZOFI steps into the next instruction to modify the register bit so that it doesn't get overwritten. Either the execution leads to completion or gets interrupted by a signal. In the case of an infinite loop, ZOFI sets up an alarm to receive a signal if the binary gets stuck. In either case, the tool compares the outcomes of the multiple test runs with the golden run.

ZOFI offers a variety of optional arguments to fine-tune a fault simulation experiment. The user can inject faults into specific registers which are read or written by instructions, explicitly or implicitly accessed, or an instruction pointer. Faults can also be injected into particular bits (of specific registers).

ZOFI does not come with a default example but takes any x86_64 Linux binary. We use /usr/bin/seq as the binary and run a fault simulation campaign of 100 experiments. ZOFI displays the time taken for the golden and test runs, respectively. On our machine, ZOFI takes 55.015 seconds to complete the execution of both golden and test runs to report 74 faults. It reported the number and the percentage of masked instructions[6], instructions that led to exceptions, instructions that led to infinite execution, and corrupted instructions. We also ran an implementation of a counter which took ZOFI 26.566 seconds to report 82 faults. ZOFI consumes about 50% memory while its CPU usage is varied.

Its lack of cycle accuracy facilitates maximum speed while keeping up the accuracy of statistical analysis. ZOFI simulates faults on binaries at native speed. It provides flexibility in choosing whether or not the user would want to inject faults into the system's library code. It provides users with different arguments making the tool customizable, handles binaries protected by error detection techniques, and can run multiple concurrent test runs depending upon the number of threads of the CPU. It has a built-in tracking system for workload executions, output checking and statistics collection. ZOFI provides ample optional arguments to users to fine-tune their search for exploitable faults. It lets us check for faults even in the case of error detection mechanism protected implementations.

[6] by masked instructions, the authors mean instructions that are not vulnerable to fault effects

It provides options for enabling debugging. It lets the user choose particular bits in particular registers to inject faults as well as the timstamp after which faults should be injected. It lets us determine the number of test runs as well as the number of faults to be injected per test run. Depending upon user's choice, outputs can be obtained in CSV file and Moufoplot format apart from having it on the console. We can decide the maximum number of fault injection attempts and the degree of parallelization.

However, there are downsides to ZOFI. Due to its timing-based design, it fails to function correctly if the workload runs for a variable or short time as it attempts to inject faults after the binary has completed its execution. Hence, executing the binary needs to take at least 0.5 seconds. If the binary shows unexpected behaviour when stopped with a signal, ZOFI cannot analyze it since it uses a signal to pause test runs. ZOFI cannot guarantee micro-architectural accuracy because it's limited to the state provided by the instruction set architecture (ISA).

### 5.3 ARMORY

**ARMORY** [12], a fully automated open-source hardware emulation framework for exhaustive fault simulation of the ARM-M binaries, simulates all exploitable arbitrary (and customizable) fault combinations (including higher-order faults) while automatically utilizing all the available CPU cores. Using various fault models, it efficiently scans a compiled binary for potential weaknesses against arbitrary (including multivariate) fault combinations. It demonstrates that considering machine code for fault injection analysis instead of code written in comparatively high-level language could prevent overlooking specific faults that don't present themselves in higher levels of abstraction.

ARMORY is based on an efficient instruction-accurate open-source emulator for ARMv6-M, ARMv7-M and ARMv7-EM architectures, named M-ulator. M-ulator can work according to a specific instruction set architecture and handle faulty assembly instructions.

ARMORY takes as input an M-ulator instance (loaded with binary), a set of fault models to inject, an exploitability model (a set of conditions which determines if a specific fault could be considered exploitable if met) and halting points (addresses where the exploitability is to be evaluated). It outputs the fault combinations along with fault tracers. It considers a total of 24 fault models, both instruction-level (permanent or transient) and register-level (permanent or transient or active until overwrite) - instruction skipping, faults on flash (program memory), faults on RAM (data memory), faulting operand registers, faulting addresses, diverting control flow, or instruction replacement. ARMORY uses an optimized fault simulation strategy by executing a dry run first, followed by fault simulation until the next fault injection point. M-ulator runs a fault-free simulation at first until the execution of the binary gets completed or the supplied timeout is reached, thus providing the sequence of executed instructions and used registers. This sequence gives all the injection points in order. After doing so, M-ulator continues until the next injection point, and the current state is stored

as backup. This saves emulation time by eliminating the need to start from the beginning for each fault. Then the fault model is applied. On reaching one of the halting points, it checks the exploitability model to determine whether the fault encountered is to be added to the list of exploitable faults. If no such halting point is reached or an invalid instruction is met, then the M-ulator's state is stored. However, this might not be enough when dealing with multivariate fault simulation. So, in the case of higher-order fault simulation, even if the current fault combination isn't found to be exploitable, the tool recursively runs the current state with the next fault model. Hence, the following fault model starts with M-ulator where a specific fault combination has already been applied. Also, in the case of multivariate faults, the order of faults holds significance. ARMORY attempts to optimize using M-ulator backups and efficient multicore support.

ARMORY comes with three default examples; one inserted faults at specific addresses, another was an AES and finally, a secure bootloader. ARMORY displays the details of its version, compiler, linker for ARMORY and M-ulator, and run-time dependency threads. It simulates all 24 kinds of fault models using threads 16 times and provides a comprehensive report which includes the fault models, number of threads used, elapsed time for each type of fault injected, number of faults exploited and number of faults injected, along with the position and time-stamp of exploited faults (if any), the assembly instruction where the fault could be injected, the total time taken and the total number of exploitable faults. ARMORY took 0.979 seconds, 24:38.730 minutes and 24:46:39.962 hours to exhaustively simulate and report 24, 823192 and 1124 faults respectively. It provides outputs both on the console and as a log file which helps in future referencing. We chose an implementation of a counter to run on ARMORY. We defined the directory for storing temporary data, the path to build the ARM binary, and disassembly to accumulate the results, start the fault injection simulation, and clear the temporary data. We also defined the start and halt symbols for the fault simulation. For simulating faults on the binary of the counter implmentation, ARMORY took 1.746 seconds to report 2 faults. ARMORY consumes about 50% memory while utilizing the entire CPU resources.

Besides being highly customizable, it offers fault injection simulation for multivariate faults. ARMORY automatically utilizes all the available CPU cores. M-ulator, explicitly designed for fault simulation, outperforms the Unicorn emulator and harbours the ability to handle incorrect assembly code, unlike other emulators. ARMORY focusses on lower abstraction levels for fault simulation binaries of ARM-M since applying isloated fault models on higher abstraction levels overlook exploitable faults.

ARMORY also comes with a few limitations, mainly because of M-ulator. M-ulator doesn't support certain ARMv6-M instructions [7]. Even though ARMORY exhaustively simulates faults, it fails to decide if a fault model is feasible in reality since every fault model is not equally probable in the real world.

[7] such as MSR (move to special register from ARM register), MRS (move to register from the special register), CPS (change processor state), SVC (supervisor call instruction)

### 5.4 ARCHIE

**ARCHIE** (ARCHitecture-Independent Evaluation) [11] is an open-source, automated, QEMU-based fault simulation tool for analyzing transient and permanent instruction and data faults in RAM, flash and processor registers of embedded devices. QEMU is used as the underlying emulator because it is generic, open-source, operating system independent, and supports numerous architectures. ARCHIE can autonomously execute a user-defined fault campaign. It supports four fault models. The Set 0 and Set 1 fault models replaces the bits to 0 and 1 respectively while the toggle fault model switches the bits represented by the fault mask. The overwrite fault model ensures that an instruction could be skipped.

The underlying emulator, QEMU, replicates the architecture of the target hardware. The controller script of ARCHIE takes the inputs, the compiled binary, QEMU configuration and fault configuration, to launch several parallel processes. Each of these parallel processes handles one QEMU instance one of the fault models from the fault configuration to run their experiment independently. Once these processes end their task, they collect all the results to store in a HDF5 file. The number of parallel processes can be set by the user.

For fault simulation on its default example, which implemented the blinking of LEDs on an STM32F0DISCOVERY board, ARCHIE displays the version, the fault campaign, timestamps, system RAM consumed, the worker number (parallelization) and the assembly instruction where the fault injection was simulated as it went along, checking for exploitable faults. It is equipped with debugging functionality. AES was another default example that came with ARCHIE. AES's fault configuration contained 4 faults which took ARCHIE 2:06:58.019 hours to simulate. We executed ARCHIE on an implementation that led to the blinking of LEDs on a STM32VLDISCOVERY board. We defined the QEMU configuration, which comprised the path to the QEMU build and the compiled binary, as well as the name of the machine, i.e., STM32VLDISCOVERY. For the fault campaign, we determined the starting and ending address, the fault address, type, model, livespan, mask and trigger address, and counter with reference to the assembly instructions of the compiled binary. In effect, ARCHIE does not exhaustively or randomly simulate the entire space. So, it can't be categorised as either deterministic or non-deterministic. It checks whether our hypothesis regarding the faults turns out to be correct or not. For its default example, with 4 faults defined, it took 6:16.714 mins for ARCHIE to execute the simulation. For the fault configuration on STM32VLDISCOVERY board, we set 1 fault to be simulated. ARCHIE reported the results of the fault simulation in 1:1.414 minutes. ARCHIE uses both terminal and log files to record its findings. It uses up the entire memory and varied percentages of the CPU. Since, ARCHIE takes up the entire RAM, it causes the kernel to kill or terminate the execution of the tool. For a few executions of these mentioned binaries, the process was killed by the kernel.

ARCHIE has a few limitations. The golden run stores the content of only ARM registers instead of storing multiple register dumps for numerous points

in time. The golden run stores memory dumps for specific locations and lengths instead of multiple memory dumps. Register dumps are not possible - only memory dumps are implemented. With overwrite fault model, up to only 16 bytes can be overwritten. While running the tool on our virtual machine, it has been seen that it consumes almost the entire memory and CPU and causes the virtual machine to get stuck.

**Table 4.** Inputs/Outputs for FiSim, ZOFI, ARMORY and ARCHIE

| Tools | Inputs | Outputs |
|---|---|---|
| Riscure FiSim (Demo Edition) | Binary in high-level code | Fault-list in assembly code redirecting to program locations |
| ZOFI | Compiled binary | 1. Timing of golden run 2. Numbers, percentages of masked and corrupted instructions, instruction resulting in exceptions and infinite loops |
| ARMORY | 1. Binary in high-level code 2. Start symbol 3. Halt symbol Path to 4. directory for temporary data 5. build ARM binary 6. dissasembly to store results 7. start emulation 8. clean up temporary data | 1. Total number of faults 2. Number of faults injected 3. Type, number, elapsed time, position, timestamp of each fault exploited 4. Number of threads used 5. Faulted instruction in assembly code 6. Affected instruction, register, bit or byte |
| ARCHIE | 1. Compiled binary 2. QEMU configuration 3. Fault configuration | Detailed working (for debugging) |

## 6 Discussion

We ran all four fault simulation tools on their default example(s) followed by executing each of them on an implementation of our choice.

There is a trade-off between user-friendliness and convenience vs usability of the fault simulation tool. FiSim and ZOFI are user-friendly and convenient to use. However, to analyze a binary, it would require the binary to include certain functions already defined in FiSim. If a binary does not follow the program structure of a secure bootloader, then one needs to re-write the engine too. Then it is possible to cross-compile the binary with the respective GNU ARM Embedded Toolchain before it can be made ready for simulation in its GUI. Plugging in the inputs for ARMORY (Table 4) is sufficient to execute it on a binary. In contrast, for ARCHIE, the user needs to determine the fault configuration from the assembly code of the binary. ARCHIE simulates and verifies our guesses with respect to the fault configuration. Working with ARCHIE, requires one to be acquainted with QEMU to decide the implementation based on the machines QEMU emulates and if one would like to provide additional arguments. Only the ARM architecture of QEMU comes pre-built. So, for using other architectures, the user needs to build it. One must cross-compile and comprehend the assembly generated well enough to decide the exact locations to inject faults. So, ARCHIE is better suited for an experienced user who is accustomed to dealing with memory locations, storage, and errors like hard faults. Also, ARCHIE writes all the generated data together at the end of each experiment which consumes a lot of RAM memory and sometimes causes the kernel to terminate or kill the process.

For reasons mentioned above, FiSim, being a prototype, isn't nearly as flexible as the other tools under comparison. Both ZOFI and ARMORY provides us with ample optional arguments to fine-tune the fault simulations. In the case of a multi-core system, ZOFI can run parallel jobs and limits the number of parallel jobs if the parameter is set mistakenly when the system doesn't support it. However, the user needs to provide ZOFI with a binary whose execution time isn't too small (0.05 seconds), or it will not return correct results. The execution of ARCHIE depends entirely on the expertise of the user.

ARMORY supports 24 fault models (Table 1) followed by that of ARCHIE which in turn is followed by ZOFI. Fisim supports the least with 2 fault models.

On the console, some tools provide helpful messages along with the details of injected faults to assist in debugging. However, unlike the other three tools, ZOFI refrains from providing the fault address and the respective instruction.

When it comes to speed, ZOFI is the fastest since it runs almost as fast as a native run. It is closely followed by FiSim. However, as we can see from Table 3 ARMORY and ARCHIE can take quite a while to complete their executions. For ARMORY, we can reduce the range by modifying the start and the halt symbols to decrease its execution time. For ARCHIE, we can reduce the scope by adjusting the start and end addresses and reducing the number of faults to be injected.

## 7 Conclusion

In our attempt to evaluate four open-source fault simulation tools, we had the opportunity to compare FiSim, ZOFI, ARMORY and ARCHIE objectively. They

**Table 5.** Comparison among FiSim, ZOFI, ARMORY and ARCHIE

| Tools | Use of CPU cores & Memory | Granularity | Injection Accuracy | Access to Micro-architecture |
|---|---|---|---|---|
| Riscure FiSim (Demo Edition) | Automatically utilizes all available CPU cores About 50% of memory | Instruction | Medium | No |
| ZOFI | Automatically utilizes all available CPU cores About 50% of memory | Instruction | Low | No |
| ARMORY | Automatically utilizes all available CPU cores More than 50% of memory | Instruction | Medium | No |
| ARCHIE | Automatically utilizes all available CPU cores About 100% of memory | Instruction | Medium | No |

employ different techniques to navigate faults and present varying difficulty levels to the user.

FiSim could be the simplest to use if only its engine were written in a way to incorporate different kinds of implementations. Barring this drawback, if the user intends to figure out fault injection locations in implementations similar to that of a secure bootloader, FiSim would be the tool of choice. However, it reports faults deterministically, which doesn't conform with the concept of fault injections. However, FiSim is one of the most elegant (equipped with GUI) tools if we re-write its engine according to our needs.

ZOFI is the simplest because it just takes the binary and the number of test runs. However, it supports binaries that run on x86_64 Linux. Though there are several arguments to fine-tune the search for faults, defining them is optional. As long as the golden run takes more than 0.5 seconds to execute, ZOFI would return valid results. It tends to produce only the number of faulted instructions sans any details. So, the tool doesn't provide any address pointing to the fault. This means it could be tedious to make an implementation fault attack resistant since one would need to run it through ZOFI numerous times. Furthermore, lack of addresses where faults are injected could make it relatively harder to rectify the faulty locations. Though, ZOFI's non-deterministic nature captures the essence of fault injection.

If the user is interested in an ARMv6M, ARMv7M or ARM7EM implementation or any other implementation that could be efficiently modified to one, then ARMORY would be the fault simulation tool to pick. Apart from being restricted to the ARM architecture, there isn't much downside. However, unlike

other tools based upon an existing emulator, ARMORY comes with an emulator of its own, M-ulator. Naturally, there are pros and cons to both sides. Built upon an existing emulator gets the tools more confidence, acceptance, and access to all the architectures that the emulator can emulate. Building one's emulator could be advantageous in designing it per target, making it more compatible with the fault injection tool.

Unlike the other fault simulation tools, which inspect the space for faults, ARCHIE only confirms if the user's estimate regarding the faults is accurate. It supports all the architectures that QEMU emulates. Though if that architecture isn't ARM, then the user would need to build it before executing ARCHIE, as only the ARM architecture of QEMU is pre-built with the tool. Also, it uses an older version of QEMU and remains to be seen if it is forward compatible with the current QEMU version.

The choice of fault simulation tool depends on the user's purpose. If the implementation follows the program structure of a bootloader, and the user cares about first-order faults, then FiSim works best. If it's a x86_64 Linux binary and the user doesn't require the fault locations then they can prefer ZOFI. In case it's an ARM-M binary, ARMORY is the tool of choice. If the user is experienced enough to estimate fault configurations and has ample memory resources, then going for ARCHIE is preferable since it provides detailed working for debugging purposes.

## References


1. The reset glitch hack : A new exploit on xbox 360 [ en ], http://www.logic-sunrise.com/news-341321-the-reset-glitch-hack-a-new-exploit-on-xbox-360-en.html
2. How the apple airtags were hacked (May 2021), https://www.youtube.com/watch?v=_E0PWQvW-14
3. Arribas, V., Wegener, F., Moradi, A., Nikova, S.: Cryptographic fault diagnosis using verfi. Cryptology ePrint Archive, Paper 2019/1312 (2019), https://eprint.iacr.org/2019/1312, https://eprint.iacr.org/2019/1312
4. Becker, M., Baldin, D., Kuznik, C., Joy, M.M., Xie, T., Mueller, W.: Xemu: An efficient qemu based binary mutation testing framework for embedded software. In: Proceedings of the Tenth ACM International Conference on Embedded Software. p. 33–42. EMSOFT '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2380356.2380368, https://doi.org/10.1145/2380356.2380368
5. BlackHatOfficialYT: Minimum failure - stealing bitcoins with electromagnetic fault injection (Jan 2020), https://www.youtube.com/watch?v=WOMAOGOvXnA
6. Bosio, A., Natale, G.D.: Lifting: A flexible open-source fault simulator. In: 2008 17th Asian Test Symposium. pp. 35–40 (2008). https://doi.org/10.1109/ATS.2008.17
7. Burchard, J., Gay, M., Ekossono, A.S.M., Horáček, J., Becker, B., Schubert, T., Kreuzer, M., Polian, I.: Autofault: Towards automatic construction of algebraic fault attacks. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). pp. 65–72 (2017). https://doi.org/10.1109/FDTC.2017.13

8. Chyłek, S., Goliszewski, M.: Qemu-based fault injection framework. Studia Informatica **33**, 25–42 (01 2012)
9. Ferraretto, D., Pravadelli, G.: Efficient fault injection in qemu. In: 2015 16th Latin-American Test Symposium (LATS). pp. 1–6 (2015). https://doi.org/10.1109/LATW.2015.7102401
10. Grycel, J., Schaumont, P.: Simplifi: Hardware simulation of embedded software fault attacks. Cryptography **5**(2) (2021). https://doi.org/10.3390/cryptography5020015, https://www.mdpi.com/2410-387X/5/2/15
11. Hauschild, F., Garb, K., Auer, L., Selmke, B., Obermaier, J.: Archie: A qemu-based framework for architecture-independent evaluation of faults. In: 2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC). pp. 20–30 (2021). https://doi.org/10.1109/FDTC53659.2021.00013
12. Hoffmann, M., Schellenberg, F., Paar, C.: Armory: Fully automated and exhaustive fault simulation on arm-m binaries. IEEE Transactions on Information Forensics and Security **16**, 1058–1073 (2021). https://doi.org/10.1109/TIFS.2020.3027143
13. Jenn, E., Arlat, J., Rimen, M., Ohlsson, J., Karlsson, J.: Fault injection into vhdl models: the mefisto tool. In: Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing. pp. 66–75 (1994). https://doi.org/10.1109/FTCS.1994.315656
14. K, K., Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: Feds: Comprehensive fault attack exploitability detection for software implementations of block ciphers. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(2), 272–299 (2020). https://doi.org/https://doi.org/10.13154/tches.v2020.i2.272-299, https://tches.iacr.org/index.php/TCHES/article/view/8552
15. Khanna, P., Rebeiro, C., Hazra, A.: Xfc: A framework for exploitable fault characterization in block ciphers. In: 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (2017). https://doi.org/10.1145/3061639.3062340
16. LimitedResults: Enter the efm32 gecko (Jun 2021), https://limitedresults.com/2021/06/enter-the-efm32-gecko/
17. Nasahl, P., Osorio, M., Vogel, P., Schaffner, M., Trippel, T., Rizzo, D., Mangard, S.: Synfi: Pre-silicon fault analysis of an open-source secure element (2022). https://doi.org/10.48550/ARXIV.2205.04775, https://arxiv.org/abs/2205.04775
18. Porpodas, V.: ZOFI: zero-overhead fault injection tool for fast transient fault coverage analysis. CoRR **abs/1906.09390** (2019), http://arxiv.org/abs/1906.09390
19. Richter-Brockmann, J., Rezaei Shahmirzadi, A., Sasdrich, P., Moradi, A., Güneysu, T.: FIVER – robust verification of countermeasures against fault injections. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(4), 447–473 (Aug 2021). https://doi.org/10.46586/tches.v2021.i4.447-473, artifact available at https://artifacts.iacr.org/tches/2021/a16
20. Riscure: Riscure/fisim: An open-source deterministic fault attack simulator prototype, https://github.com/Riscure/FiSim
21. Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: Safari: Automatic synthesis of fault-attack resistant block cipher implementations. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(4), 752–765 (2020). https://doi.org/10.1109/TCAD.2019.2897629
22. Saha, S., Alam, M., Bag, A., Mukhopadhyay, D., Dasgupta, P.: Leakage assessment in fault attacks: A deep learning perspective. Cryptology ePrint Archive, Paper 2020/306 (2020), https://eprint.iacr.org/2020/306, https://eprint.iacr.org/2020/306

23. Saha, S., Kumar, S.N., Patranabis, S., Mukhopadhyay, D., Dasgupta, P.: Alafa: Automatic leakage assessment for fault attack countermeasures. In: 2019 56th ACM/IEEE Design Automation Conference (DAC). pp. 1–6 (2019)
24. Saha, S., Mukhopadhyay, D., Dasgupta, P.: Expfault: An automated framework for exploitable fault characterization in block ciphers (revised version). Cryptology ePrint Archive, Paper 2018/295 (2018). https://doi.org/10.13154/tches.v2018.i2.242-276, https://eprint.iacr.org/2018/295, https://eprint.iacr.org/2018/295
25. Simevski, A., Kraemer, R., Krstic, M.: Automated integration of fault injection into the asic design flow. In: 2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS). pp. 255–260 (2013). https://doi.org/10.1109/DFT.2013.6653615
26. Srivastava, M., SLPSK, P., Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: Solomon: An automated framework for detecting fault attack vulnerabilities in hardware. In: Design, Automation, and Test in Europe Conference Exhibition (DATE). pp. 310–313. DATE, IEEE (2020). https://doi.org/10.23919/DATE48585.2020.9116380, https://ieeexplore.ieee.org/document/9116380