SoK: Assisted Fault Simulation
Existing Challenges and Opportunities Offered by AI

Asmita Adhikary and Ileana Buhan

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

Abstract. Fault injection attacks have caused implementations to behave unexpectedly, resulting in a spectacular bypass of security features and even the extraction of cryptographic keys. Clearly, developers want to ensure the robustness of the software against faults and eliminate production weaknesses that could lead to exploitation. Several fault simulators have been released that promise cost-effective evaluations against fault attacks. In this paper, we set out to discover how suitable such tools are, for a developer who wishes to create robust software against fault attacks. We found four *open-source* fault simulators that employ different techniques to navigate faults, which we objectively compare and discuss their benefits and drawbacks. Unfortunately, none of the four open-source fault simulators employ artificial intelligence (AI) techniques. However, AI was successfully applied to improve the fault simulation of cryptographic algorithms, though none of these tools is open source. We suggest improvements to open-source fault simulators inspired by the AI techniques used by cryptographic fault simulators.

Keywords: Fault simulation · Open-source · Software · AI

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 [22], for example, through clock glitches, where short glitches are inserted into the clock signal, which may cause timing violations or voltage glitches, where the device is supplied with power outside the range of values specified in the datasheet. *Fault injection* is the deliberate action of modifying or skipping the intended flow of operations that could result in the software acting unexpectedly. A successful fault injection attack may allow unauthorized individuals to access off-limits memory locations or bypass necessary authorization conditions that could crash the system or cause it to behave unexpectedly, leading to dire consequences.

Fault injection attacks in vehicle immobilizer systems [37], or key recovery of Playstation Vita AES-256 [20] provide enough incentive to prevent further attacks by injection of faults. Unauthorized access to a Linux operating system [34] or exploring embedded systems [35,36] by exploiting faults further proves the disastrous effects of fault injection attacks.

Assisted fault simulation automates detecting and testing vulnerabilities in a given program's control flow or due to data modification. A *fault simulator* is a software tool that simulates the effects of faults or errors in digital circuits or software programs. When targeting software, a fault simulator will replicate (either by simulating or emulating[1]) the underlying architecture on which the program will run. A user-specified *fault model* specifies the parameters for the fault simulator to look for exploitable locations. The target of the fault simulator can be any part of the implementation, hardware, or software. To determine whether the target is vulnerable to faults, a *test case* modifies the target according to one instance of the fault model and executes the target. If the target behaves as expected, the tested location is not vulnerable. Alternatively, the target is vulnerable, and the user can determine what caused the fault.

One of the benefits of using a fault simulator is the possibility of performing a root cause analysis. When performing fault injection on targets, it is impossible to determine what caused the fault. A fault simulator allows users to determine the cause of a successful fault and harden the implementation without needing a physical target and expensive tools. Additionally, using fault simulators may be less complex and expensive than testing for faults using real fault injection tooling on an actual target. Given the appeal of fault simulators, we sought to investigate state-of-the-art open-source fault simulators. The target audience for this paper is developers of *general-purpose software* and not specifically cryptographic implementations. Although many fault simulators [6,18,31,29,26,1,17,33,24,28,27,16,14,5] can verify the implementations of cryptographic algorithms, we exclude them in this work because there is already an overview of such tools [4]. We found four fault simulators in the public domain suitable for such a developer, namely FiSim [25], ZOFI [23], ARMORY [13] and ARCHIE [12].

In this paper, we discuss what features existing tools offer, which use cases they cover, and how easy it is to adapt them to different examples. AI was successfully applied to improve the fault simulation of cryptographic algorithms [4]. Hence, we explore the use cases where AI was successfully applied and discuss the opportunities to apply the techniques employed by the latter to optimize the former. A fault simulator must satisfy the following conditions to provide a significant advantage over performing fault injection on an actual target:

1. It must be fast enough to cover as many test cases as possible in a reasonable amount of time.
2. It needs to be scalable, as it should handle smaller code snippets and full-fledged implementations of real-world applications.
3. It should offer interpretability so that a developer can use the results to harden the implementation.

Not all *vulnerable locations* are *exploitable* as attacking a vulnerable location may or may not result in an exploitable fault. For example, the fifth-round byte fault in AES is not exploitable, whereas the eighth- or ninth-round byte fault

[1] The terms, simulation, and emulation, have been used interchangeably.

in AES is exploitable [31]. Therefore, there might be innumerable vulnerable locations, but not all of them will be exploitable. The goal of a fault simulator is to find exploitable locations.

Contribution. In this paper, we offer the following contributions:

1. We propose a grammar to express fault models that can be used to compare the capabilities of each tool at a glance in Section 3.
2. We define a set of parameters to compare the tools to help a prospective user quickly decide which tools best suit his use case in Section 4.
3. We objectively compare existing open-source fault simulators designed for general-purpose software while discussing their operation and features in Section 5.
4. We present challenges of existing fault simulators in Section 6.
5. We suggest optimizations to improve existing fault simulators with AI techniques in Section 7.

Paper organization. The remainder of the paper is organized as follows. Section 2 presents the development of fault simulator over the years. Section 3 outlines our proposed grammar to express fault models. Section 4 describes the criteria we chose to evaluate the four fault simulators. Section 5 details the experimental setup along with the fault simulators and their functioning and features. Section 6 identifies the challenges of using the existing fault simulators, followed by suggestions for improvements using AI techniques in Section 7. Finally, we conclude in Section 8.

2 Background

There are surprisingly many tools available to perform fault simulations. We first divide existing tools according to the type of implementations, *hardware circuits* vs. *software programs*. The intended application for the verified target can be divided into *cryptographic implementations* vs. *general-purpose software*. Lastly, we consider whether the tools are open source. When discussing related work, we mention fault simulators that we could not execute. Due to the large number of tools available [15,3,8,2,32,10,25,23,11,13,12,21], we make a representative selection of simulators available for general-purpose software.

Fault simulation on hardware circuits. The first general-purpose tool published to test circuit fault resistance is MEFISTO (Multilevel Error/Fault Injection Simulation TOol) [15], a proprietary tool. 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 models. It also estimates the possible coverage with the given fault-tolerance mechanisms. LIFTING (LIRMM Fault Simulator) [3] is an open source, object-oriented 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 (SEU). The simulator checks

if the design meets the expectations of the functional specifications and introduces fault injections for stuck-at and SEU fault models. [32] has proposed an automated integration of fault injection into the ASIC design flow.

Fault simulation on software programs. All fault simulators in this category use QEMU as the underlying emulator. QEFI [8] focused on the ARM architecture, executes a system-wide and kernel-based fault simulation to check its susceptibility to faults. The next framework is XEMU [2], a language and compiler-independent tool that performs efficient mutation-based testing of software binaries by injecting mutations at run-time using dynamic code translation. It uses QEMU in user mode, 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 injection. XEMU approaches 100% accuracy for test quality metrics compared to source code instrumentation. The CFG offers a speed-up of up to 100 - 1000 times with a GDB/ARMulator. EQEFI [10] is automatic and non-intrusive, minimizing the effect of fault simulation on emulator performance by simulating the presence of permanent, intermittent, and transient faults in the CPU registers of both RISC and CISC architectures.

AI-based fault simulators. Cryptographic fault simulators turned to AI to help cover as many test cases as possible. Techniques vary from data mining to machine learning and deep learning. The first of its kind, ExpFault [31], used data mining to detect exploitable locations (for differential fault analysis) in block-cipher implementations. ExpFault mines distinguishers from fault simulation data using association rule mining to group frequent items. For the recovery of the key after the implementation of the cipher, DL-FALAT [27] uses the deep learning-based leakage detection test based on the principle of non-interference [9] as an improvement to the t-test used in ALAFA [29]. To construct a dataset for the machine learning algorithm, [28] uses a SAT solver to retrieve the key from a few instances of the cipher implementation. The dataset is used to train the machine learning algorithm to predict if there exist any other fault instances in the same cipher implementation. Different algorithms will use different datasets and models. Carpi et al. [7] were the first to use genetic algorithms to improve parameter selection for fault injection. Their solution to narrow down the space of exploitable locations from the vulnerable locations could be beneficial. Krcek et al. [19] utilizes genetic algorithms and machine learning to develop a solution that allows them to find more exploitable faults compared to using either of the approaches.

3 A Taxonomy of Fault Models

Due to the lack of a taxonomy regarding fault models, different fault simulators use different terminologies to refer to the supported fault models. For example, ZOFI supports a single-event upset fault model in which it attempts to perform a bit-flip in a register whereas ARMORY supports 24 fault models. ARCHIE supports four fault models. The set 0 and set 1 fault models replace the bits

Table 1. Fault Models with Abbreviations

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

to 0 and 1, respectively, while the toggle fault model switches the bits represented by the fault mask. The overwrite fault model ensures that instructions can be skipped. However, the description of the fault model lacks clarity and it remains unspecified whether the fault is to affect an instruction (I), data (D) or address (A). Having a taxonomy helps the fault simulators present the supported fault models in a concise manner. As seen from the Table 1, set 0, set 1, toggle and overwrite fault models correspond to the actions, clear (C), set (S), flip/toggle (G) and skip/NOP (instruction) (N) respectively. However, different fault simulators using different terminologies to refer to the same fault model adds ambiguity. Moreover, the duration of the fault manifestation and its effects remain undefined.

We found no unified view to describe fault models when comparing existing tools, so we proposed a solution that allows us to describe the fault models for the tools we compared.

Table 1 shows our solution to describe the characteristics of a fault. The first parameter, *how*, describes the duration during which the faults manifest. Unlike *transient faults*, which affect only the current run of the target program, *permanent faults* affect the current and all future executions. *Until-overwrite* refers to the faulting of a register or any other memory location, the effects of which become nullified once its value is overwritten. The parameters *who* and *resolution* describe which implementation fragment is affected. Finally, each fault model is described by an *action* with the help of which the fault is injected.

For example, FiSim[25] supports two fault models: *transient instruction skipping* ([T][I][N]) and *transient instruction bit flips*([[T][I][b][G]]). While the former fault model skips the subsequent instruction, the latter flips the value of a bit corresponding to an instruction for a particular execution of the program. The combination in *fault models* does not imply that the tools support all possible combinations of fault models.

4 Criteria for evaluating the fault simulators

This section discusses our criteria for evaluating the four open-source fault simulators. From a developer's perspective, these criteria aim to provide quick insight into the working of each tool to help decide which tool to use. Figure 1 illustrates the various parameters that we use to evaluate the fault simulators.

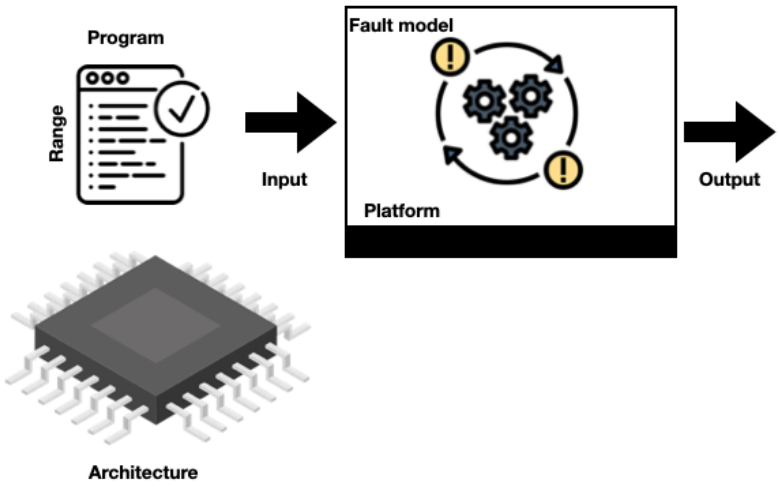

Fig. 1. Illustration of our criteria for evaluating fault simulators

Figure 1 portrays the inputs needed to run the target program on the fault simulators. Foremost is the implementation itself, which can be cross-compiled, as suggested by the OS and the architecture. Depending on the fault simulator, the developer can specify the range of the implementation to be considered when the fault simulator attempts to induce faults into it. The "program" is supposed to run on the "architecture". Hence architecture can also be looked at as an input. Depending on the inherent design of the fault simulator, certain fault models will be executed either deterministically or randomly. The "platform" helps users decide on the host system to run the fault simulator.

Table 2 is a concise overview of the parameters we selected for comparison. *Platform* informs us about the underlying simulation or emulation engines. The *OS* informs us about the operating system for which the fault simulator will run and compatible cross-compilers. *Architecture* describes the supported hardware device on which the execution of the software can be replicated. *Range* specifies whether a tool is *exhaustive*(E) and will consider the entire implementation or is *user-defined*(U), and the user can specify the range of instructions. At first glance, tools that support an exhaustive search for vulnerable locations might seem tempting, but could hinder handling an implementation that consumes substantial resources. On the contrary, the tools supporting user-defined range help us fine-tune as per requirements and might prove beneficial.

Coverage determines whether the injected faults are *deterministic* (D), where multiple runs of the fault simulation experiments will always produce the same results, or *random*(R), where multiple runs of the fault simulation experiments will produce different results. Deterministic faults do not capture the essence

Table 2. Evaluation Criteria: E=Exhaustive, U=User-defined, D=Deterministic, R=Random, Refer to Table 1 for fault model's grammar

Tool	Platform	OS	Architecture	Range	Coverage	Fault Models	Memory Consumption
FiSim	Unicorn	Windows (Pre-built) Linux & Mac	ARM	E	D	[T][I][b][G,N]	≈ 50%
ZOFI	Native hardware	Linux	x86_64	E	R	[T][A][b,R][G]	≈ 50%
ARMORY	M-ulator	Linux	ARMv6M, ARMv7M, ARMv7EM	U	D	[*][I][*][*]	> 50%
ARCHIE	QEMU	Linux	Depends on QEMU	U	D	[P,T][I,D][b,R] [C,S,G,F]	≈ 100%

of fault injections. Still, they are readily reproducible, whereas random faults function exactly as fault injections in an actual setup. However, the results may not be so easily reproducible. One of the most important features of a fault simulator is the supported *fault models* (Section 3).

The parameter *memory consumption* tells us how much memory the simulator occupies while working. Using significant memory resources could cause the system to terminate the simulation forcibly.

5 Experimental Setup

We installed ZOFI (version 0.9.7)[2] [23], ARMORY[3] [13] and ARCHIE[4] [12] 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, 1 physical processor, 8 cores, and 16 threads with 50.43 GB of RAM. We installed the ARM GNU toolchain and the Meson build system as prerequisites to run ARMORY. We ran FiSim[5] [25] on Windows 11 Pro, AMD Ryzen 5 PRO 5650U @ 2.30GHz, and 16 GB of RAM to facilitate the use of its GUI.

To compare, we first run each tool with the default example. Initially, our goal was to run the same code on all tools, but this was impossible due to differences in the platforms. So, we ran each tool with a different example. In the next section, we describe each tool in detail.

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

5.1 FiSim

Description. FiSim [25], based on the Unicorn emulator and Capstone disassembler, is a prototype of a deterministic fault attack simulator. It supports cross-platform simulation of ARM32/ARM64 architectures. Its GUI is pre-built for Windows but supports Linux and Mac OSX systems. FiSim implements two fault models, as shown in Table 2.
Operation. FiSim comes preloaded with a secure bootloader implementation. The first boot stage takes as input the implementation of the bootloader. The tool provides console output for debugging and visualizing 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 whether authentication succeeded. Then the execution trace is placed in the fault simulator. The number of test cases depends on the size of the code. For every instruction and the two supported fault models, FiSim will modify the target program according to the fault model, run the program, and compare the result of the execution with the expected output. If and when the code is vulnerable, the authentication will succeed, and the next boot stage will be executed, which signifies that the glitch bypassed the authentication. The target source code of the secure bootloader is compiled, and the resulting binary code is used for simulation. When the implementation is run, FiSim shows a list of assembly instructions.
Features. FiSim provides ways to harden the bootloader by pointing out its weaknesses and testing the effectiveness of its countermeasures. FiSim aims to find the balance between accuracy and speed. However, loading other than the default programs for simulation is not straightforward.

5.2 Zero Overhead Fault Injector

Description. ZOFI [23] is a timing-based fault simulator based on the Capstone library. Unlike the other tools, it uses native hardware (x86_64 Linux systems), to run the target program. ZOFI implements a register single-bit flip fault model.
Operation. ZOFI does not have a default example, but takes as input any x86_64 Linux binary. During the golden run, ZOFI executes the unmodified binary at native speed to measure its execution time and collect its original output. The execution time of the golden run serves as an upper bound for the fault simulation time. It helps to approximate whether the binary needs to be terminated. The results of the golden run help ZOFI compare the results of the subsequent test cases to categorize instructions as corrupted, masked[6], detected, stuck in an infinite loop, or throwing an exception. For each test case, ZOFI forks and launches a new process to run the binary, pausing 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

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

fault simulation, the execution of the binary is interrupted by a signal emitted by ZOFI. Then ZOFI modifies the state of the binary by injecting a register bit-flip. Depending on whether the registers are read or written by an instruction, ZOFI behaves differently. In the case of the registers 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 is interrupted by a signal. In case of infinite loops, ZOFI sets up an alarm to receive a signal if the binary gets stuck.
Features. ZOFI offers a variety of optional arguments to fine-tune a fault simulation experiment. Its main feature is the speed with which it can analyze a given workload. The user can specify a fault model by choosing the specific registers and the particular bits affected by a fault. 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 CPU threads. It has a built-in tracking system for workload executions, output checking, and statistics collection. ZOFI provides ample optional arguments for users to fine-tune their search for exploitable faults. ZOFI allows the user to determine the number of test cases and faults injected per test run. Depending upon the user's choice, outputs can be obtained in CSV file and Moufoplot format apart from having it on the console. The user can also decide the maximum number of fault simulation attempts and the degree of parallelization.

5.3 ARMORY

Description. ARMORY [13] is an efficient, instruction-accurate emulator for ARM-M binaries. It uses M-ulator which can work with ARMv6-M, ARMv7-M, and ARMv7-EM instruction set architectures and can handle faulty assembly instructions. ARMORY considers a total of 24 fault models (Table 2), both instruction-level (permanent, transient) and register-level (permanent, transient, active until overwrite). It determines all exploitable arbitrary (and customizable) fault combinations while automatically utilizing all available CPU cores.
Operation. ARMORY uses an optimized fault simulation strategy by executing a dry run first. For the next run, once ARMORY encounters exploitable locations, it adds it to a list of exploitable locations and instead of starting afresh, starts from the unmodified version of the implementation which was stored as a state in M-ulator, thus saving emulation time. M-ulator runs a fault-free simulation at first until the execution of the binary is completed or the supplied timeout is reached, thus providing the sequence of executed instructions and used registers. This sequence gives all injection points in order. After doing so, the M-ulator continues until the next injection point, and the current state is stored as a backup. This approach 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, ARMORY 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 restored. 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 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, for multivariate faults, the order of faults holds significance.
Features. As advantages, we can enumerate that it offers fault simulation for multivariate faults and is highly customizable. ARMORY provides outputs both on the console and as a log file which helps in future referencing. ARMORY attempts to optimize using M-ulator backups and efficient multicore support. It automatically utilizes all the available CPU cores. M-ulator, explicitly designed for fault simulation, outperforms the Unicorn emulator and harbors the ability to handle incorrect assembly code, unlike other emulators. It focuses on lower abstraction levels for fault simulation binaries of ARM-M since applying isolated fault models on higher abstraction levels overlooks exploitable faults.

5.4 ARCHitecture-Independent Evaluation

Description. ARCHIE [12] is a fault simulator for analyzing any binary files which can run on a QEMU-supported architecture. ARCHIE can be used for different white-, gray-, and black-box architecture testing. The tool supports specialized microcontrollers. ARCHIE executes a user-defined fault campaign. It natively supports four fault models. ARCHIE needs the developer to input the parameters of each test case. The tool does not perform fault exploration, but tests if a fault is exploitable. Instead of requiring the developer to specify the exploitable locations, it would be helpful if ARCHIE could independently predict the possible fault locations.
Operation. The ARCHIE controller script takes inputs, compiled binary, QEMU configuration, and fault configuration to launch several parallel processes. Each of these parallel processes handles one QEMU instance and one of the fault models from the fault configuration to run their experiment independently. Once these processes have completed their task, they collect and store all the results in an HDF5 file. The user can set the number of parallel processes. Each worker launches their QEMU instance along with their unique fault configuration. In effect, ARCHIE does not exhaustively or randomly simulate the entire space. It checks whether our hypothesis regarding the faults turns out to be correct or not. It uses terminal and log files to record its findings. It uses up the entire memory and uses various percentages of the CPU. Since it takes up the whole 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.
Features. ARCHIE is equipped with debugging functionality.

5.5 Experimental Results

To compare the tools available, we recorded their execution times with different implementations. For FiSim, we considered the implementation of a password authenticator. Similarly, as with the secure bootloader implementation, the

Table 3. Experimental results. The *default examples* are depicted in italic font.

Tool	Implementation	Binary Size	Execution Time	#Successful 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 (for 100 runs)
	Counter	16KB	26.566 sec	82 (for 100 runs)
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

password authentication procedure would compare the input password with the stored password. For ZOFI and ARMORY, we chose a simple implementation of a counter-loop. We executed ARCHIE on an implementation that led to the blinking of LEDs on a STM32VLDISCOVERY board.

Table 3 shows the results of the experiments for each tool[7] that highlight the implementation size, the time required, and the number of successful faults. We noticed that memory requirement heavily influences performance. Therefore, we monitor it (in Table 2) to guide us toward selecting a host machine.

6 Limitations of Existing Fault Simulators

The Fault Exploration Problem. The fault simulators we explored employ *exhaustive search* to find exploitable faults. From a security perspective, exhaustive exploration might seem tempting. The more test cases a fault simulator executes, the more successful the tool becomes in ruling out possible fault attacks and the more trust we have in a negative outcome. However, when dealing with real-world implementations, exhaustive search falls short of expectations because the required resources depend on the implementation size, the fault models supported (see Table 1), and the time it takes to run each test case. The results reported in Table 3 only consider the single fault model, while attacks involving multiple faults are known. When multiple fault instances are considered, the space for exhaustive search will increase further. Therefore, the exhaustive search might not be the goal to aim for, but having a smart selection of test cases might be a better alternative.

[7] 24:46:39.962 hrs implies 24 hours, 46 minutes and 39.962 seconds

FiSim and ARMORY exhaustively simulate all possible faults. The examples we run in our tests are fairly small (a few KB), and testing took up to 24 hours for some examples. When adding more complex fault models, the simulation time will increase further.

Cryptographic fault simulators such as [28,14,5,14] have an advantage in this respect, since cryptographic implementations follow a structure that can be analyzed using cryptanalysis. The structured nature of a cryptographic implementation helps narrow the search space, as not all parts of the implementation are equally susceptible to faults. Therefore, approaches that use SAT solvers [28], or SMT solvers [14], or data flow graphs (DFGs) (DATAC [5], TADA [14]) are suitable for this purpose. DFGs tend to grow exponentially with the size of implementation, so targeting the entire implementation space when expressed as a DFG is not feasible. When considering general-purpose software implementations, we are left with the option of exhaustively searching for faults throughout the implementation. This quickly becomes infeasible with the increased implementation size, supported fault models, and time required to run each test case.

The need for speed. Although the fault simulators might perform satisfactorily for relatively small implementations, as seen in Table 3, how these tools will scale for real-world applications is unclear. Several optimizations have been proposed in the literature to reduce testing time. Table 3 does not serve as a benchmark for comparing the speed of the fault simulators. The same binary can not be executed on all the fault simulators due to platform differences. The binary size, execution time, and the number of successful faults, as depicted in Table 3, are inconclusive. The reason is the differences in platforms and fault simulators. For example, implementing a simple counter-loop for ZOFI and ARMORY differs. The loop in the case of ZOFI has to be made large enough so that its runtime was more than 0.05 seconds. However, ARMORY can execute smaller loops. The implementation of the secure bootloader for FiSim and ARMORY is different as the example codes came embedded with the fault simulators. ARMORY requires the developer to define the starting and halting points, and ARCHIE requires the developer to define the fault locations. The AES implementations for ARMORY and ARCHIE differ, too, since the former performs a one-byte fault attack after the MixColumns operation of round 7 and before the final SubBytes operation to reduce the key-search space while the latter performs a tenth round skip attack and a diagonal fault attack on AES.

Based on the platform used by the fault simulators, we can expect the fault simulators to behave as described henceforth. ZOFI might be the fastest tool. As it runs on the native architecture, its execution does not depend on any emulator. ZOFI pauses the binary for a random duration (with the maximum duration recorded from the golden run) to induce a fault that randomizes the fault simulation. We expect its performance to be closely followed by FiSim, which uses Unicorn, a lightweight and stripped-down variant of QEMU that offers smaller size and memory consumption as it emulates just the CPU. Since M-ulator, which underlies ARMORY, emulates only ARMv6-M, ARMv7-M, and ARMv7-EM architectures while leaving out some features, its speed is faster than

QEMU. Furthermore, the M-ulator stores an intermediate unmodified state of a binary so that ARMORY can resume testing the saved state after encountering an exploitable location instead of having to execute the binary from the very beginning. This reduces emulation time considerably. Depending on the fault specifications, ARCHIE might take the most time to complete its executions. QEMU, the platform ARCHIE uses, emulates the full system, taking up more time and memory. The time consumed by ARCHIE also depends on the fault specifications provided by the developer. The fewer the test cases, the less time is required for the simulation.

ZOFI, ARMORY and ARCHIE employ parallelization to engage in optimum usage of the system's resources and reduce the time required to run the fault simulations. ZOFI lets the developer decide or, depending on the number of available cores, decides the number of parallel threads to deploy on the fault simulations. ARMORY automatically utilizes all the available CPU cores, while ARCHIE launches independent worker threads, each of which pertains to a single QEMU instance. Finally, all the worker threads report to the main controller script, consolidating the results in a user-readable format.

Interpretability. Learning the exact instruction or data register responsible for the fault and the exact fault model is an important feature of a fault simulator. Knowing the cause will help a developer harden the implementation. For a security evaluation, the availability of debugging functionality that describes the operations of the simulator could also prove helpful. While reporting the number of successful faults is common, information on the most successful fault models is not readily available in the tools we tested. However, the lack of clarity of the results provided by simulators such as ZOFI poses a problem to the developer since it fails to provide any details of the exploitable locations. ZOFI refrains from pointing to faulted instructions and provides just the number of faults. ARMORY provides the faulted instructions along with the fault model that caused it. ARCHIE provides a debugging functionality that takes us step-by-step through its working. The lack of information on the number of test cases is another drawback of the aforementioned fault simulators. All but ARMORY are not transparent with their success percentage, i.e., number of injected faults vs. successful faults. The number of injected vs. successful faults informs us of the success percentage of the fault simulator. Due to the timing-based design, ZOFI fails to function correctly if the workload runs for a variable or a short time, as it attempts to inject faults after the binary has completed its execution, thus affecting its *accuracy*.

Realistic fault models. FiSim supports two fault models that it exhaustively applies over all possible fault locations (instructions). However, all potential faults are not equally realistic or probable in the real world. Although ARMORY supports 24 fault models, it is unclear which ones are the most probable. Although many fault models are known in the literature, no study indicates which are more *likely*. We would like to note that this is not a failure of existing simulators, but a gap of knowledge in the state of the art.

7 Opportunities of using AI Techniques

Artificial intelligence (AI) techniques and overlapping disciplines like data mining provide us with ideas to optimize the operations of fault simulators. AI models learn from a training dataset to develop and hone their search space rather than exhaustively trying all combinations. Tools such as XFC [18], SAFARI [26], FEDS [17], SOLOMON [33] and ALAFA [29] use the concept of non-interference to their advantage, while tools such as DL-FALAT [27], or ExpFault [31], or the ones presented in [30] and [28] employ deep learning and data mining, respectively, to counter the limitations outlined in Section 6. ALAFA [29] and DL-FALAT [27] attempts to detect leakage in protected block cipher implementations using the principle of non-interference to improve upon the t-test methodology without the knowledge of the cipher or the countermeasure implementation. Both of them function by observing the distributions of the faulty ciphertexts. However, unlike ALAFA [29], DL-FALAT [27] employs deep learning to enhance the leakage assessment t-test used by the former with respect to data complexity. Saha et al. [27] show that the leakage detection test needed five times less ciphertexts when employing the deep learning aproach. Unlike the t-test based leakage assessment, the deep learning based leakage detection ensures the coverage of all the necessary points in the trace for leakage detection without having the user define the order of the statistical test as well as covers the fault space reasonably better as compared to conventional approaches. This could help fault simulators in their operations. Of course, creating a representative training dataset comes with its set of challenges.

Informed Fault Exploration. Inspired by [27] we could integrate AI into the design of fault simulators by simulating simpler but structurally similar implementations. In this way, the fault simulator could exhaustively search vulnerable locations to identify exploitable locations. Executing fault simulations on numerous simplified implementations could train the model to predict faulty locations in the original target implementation. Since the original target implementation is much larger, this could be a significant improvement.

FiSim and ARMORY could apply this method to prevent solving each fault instance exhaustively and deterministically. Fault simulators attract developers because manual identification of exploitable locations is cumbersome and error-prone. However, ARCHIE requires the developer to specify the fault configurations explicitly. Hence, it could benefit from an AI model which could predict the test case for target implementations instead of the developer. This would enhance their scalability and help them closely replicate real-world scenarios by preventing deterministic exploration of faults.

Prioritizing fault models. FiSim and ARMORY simulate all the fault models described individually to arrive at a result. ARCHIE depends on the developer to specify the fault model along with the locations. So, in essence, the simulators consider all models indiscriminately or the developer is left to decide. AI models could be used to inform the choice of the parameters for the test cases as done by [7] and [19] which showed that a certain combination of parameters leads to an increase in the number of exploitable faults. A successful AI model would be

able to eliminate (to an extent) less probable fault models from the ones which occur more frequently.

Cryptographic fault simulators use SAT solvers to solve a few instances. These solutions are fed to the machine learning algorithm during its training phase. However, fault simulators pertaining to any programs could not employ similar techniques. Our suggestion would be to use exhaustive search on a few instances comprising smaller implementations of similar structure. The resulting exploitable locations could be used to build the dataset for the training phase of the AI model. The AI model could be trained to differentiate between exploitable and non-exploitable locations. It is expected that the model would predict exploitable locations for target implementations.

8 Conclusion

To evaluate fault simulators, we compared FiSim, ZOFI, ARMORY and ARCHIE objectively. They employ different techniques to navigate faults and present varying difficulty levels to a developer not used to performing fault injection attacks.

FiSim is the most intuitive to use if only its engine were written in a way to incorporate different kinds of implementations. ZOFI is the simplest tool to set up because it just takes as input the binary file and the number of test runs. 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 0.05 seconds to execute, ZOFI will return valid results. Unfortunately, it provides little information to its user. The output records the number of faulted instructions sans any details. So, the tool doesn't provide any address pointing to the fault. It could be tedious to make an implementation fault attack resistant since one would need to run it through ZOFI numerous times. Furthermore, the lack of addresses where faults are injected could make it relatively hard to rectify the faulty locations.

If the developer is interested in an ARMv6-M, ARMv7-M, or ARM7-EM implementation, then ARMORY would be the fault simulator to pick. However, unlike other tools based upon an existing emulator, ARMORY comes with an emulator of its own, the M-ulator. Naturally, there are pros and cons to both sides. Built upon a current emulator gives the tool's user more confidence, acceptance, and access to all the architectures that the platform can emulate. Building one's platform could be advantageous in designing per target, making it more compatible with the fault simulator.

Unlike the other fault simulators, which actively search for faulty locations, ARCHIE only confirms if the user's test case is exploitable. The tool offers support for all the architectures that QEMU emulates. Though if that architecture is not ARM, the user needs to build it before executing ARCHIE. Also, it uses an older version of QEMU, and it is unclear if the tool is forward-compatible with newer QEMU versions.

The choice of fault simulator depends on one's purpose. If the implementation follows the program structure of a bootloader, and one cares about first-order faults, FiSim works best. If it is an x86_64 Linux binary and the user doesn't

require the fault locations, then ZOFI might be a better choice. If it is 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.

To conclude, applying the optimizations with the help of AI techniques could enhance the fault simulators' speed and efficiency. While the deterministic nature of fault simulations might seem appealing as the results could be readily reproduced, it fails to capture the essence of fault injections. AI techniques could help in introducing a non-deterministic or probabilistic approach, which would be closer to real-life use cases. It can also help rule out fault models which aren't as equally probable as other fault models, thus, further fine-tuning the probabilistic approach of inducing faults. Conserving memory could be another incentive for employing AI in the design and development of fault simulators.

Acknowledgments. This work received funding in the framework of the NWA Cybersecurity Call with project name PROACT with project number NWA.1215.18.014, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

References

1. Arribas, V., Wegener, F., Moradi, A., Nikova, S.: Cryptographic fault diagnosis using verfi. In: 2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020. pp. 229–240. IEEE (2020)
2. Becker, M., Baldin, D., Kuznik, C., Joy, M.M., Xie, T., Müller, W.: XEMU: an efficient QEMU based binary mutation testing framework for embedded software. In: Jerraya, A., Carloni, L.P., Maraninchi, F., Regehr, J. (eds.) Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012. pp. 33–42. ACM (2012)
3. Bosio, A., Natale, G.D.: LIFTING: A flexible open-source fault simulator. In: 17th IEEE Asian Test Symposium, ATS 2008, Sapporo, Japan, November 24-27, 2008. pp. 35–40. IEEE Computer Society (2008)
4. Breier, J., Hou, X., Bhasin, S.: Automated Methods in Cryptographic Fault Analysis. Springer (2019)
5. Breier, J., Hou, X., Liu, Y.: Fault attacks made easy: Differential fault analysis automation on assembly code. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(2), 96–122 (2018)
6. Burchard, J., Gay, M., Ekossono, A.M., Horácek, 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 2017, Taipei, Taiwan, September 25, 2017. pp. 65–72. IEEE Computer Society (2017)
7. Carpi, R.B., Picek, S., Batina, L., Menarini, F., Jakobovic, D., Golub, M.: Glitch it if you can: Parameter search strategies for successful fault injection. In: Francillon, A., Rohatgi, P. (eds.) Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013.

Revised Selected Papers. Lecture Notes in Computer Science, vol. 8419, pp. 236–252. Springer (2013)
8. Chyłek, S., Goliszewski, M.: Qemu-based fault injection framework. Studia Informatica **33**, 25–42 (01 2012)
9. Clark, D., Hunt, S., Malacaria, P.: Quantified interference: Information theory and information flow. In: Workshop on Issues in the Theory of Security (WITS'04) (2004)
10. Ferraretto, D., Pravadelli, G.: Efficient fault injection in QEMU. In: 16th Latin-American Test Symposium, LATS 2015, Puerto Vallarta, Mexico, March 25-27, 2015. pp. 1–6. IEEE Computer Society (2015)
11. Grycel, J.T., Schaumont, P.: Simplifi: Hardware simulation of embedded software fault attacks. Cryptogr. **5**(2), 15 (2021)
12. Hauschild, F., Garb, K., Auer, L., Selmke, B., Obermaier, J.: ARCHIE: A qemu-based framework for architecture-independent evaluation of faults. In: 18th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2021, Milan, Italy, September 17, 2021. pp. 20–30. IEEE (2021)
13. Hoffmann, M., Schellenberg, F., Paar, C.: ARMORY: fully automated and exhaustive fault simulation on ARM-M binaries. IEEE Trans. Inf. Forensics Secur. **16**, 1058–1073 (2021)
14. Hou, X., Breier, J., Zhang, F., Liu, Y.: Fully automated differential fault analysis on software implementations of block ciphers. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(3), 1–29 (2019)
15. Jenn, E., Arlat, J., Rimén, M., Ohlsson, J., Karlsson, J.: Fault injection into VHDL models: The MEFISTO tool. In: Digest of Papers: FTCS/24, The Twenty-Fourth Annual International Symposium on Fault-Tolerant Computing, Austin, Texas, USA, June 15-17, 1994. pp. 66–75. IEEE Computer Society (1994)
16. K, K., Rebeiro, C.: Faultmeter: Quantitative fault attack assessment of block cipher software. IACR Transactions on Cryptographic Hardware and Embedded Systems **2023**(2), 212–240 (Mar 2023)
17. K., K., Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: FEDS: comprehensive fault attack exploitability detection for software implementations of block ciphers. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(2), 272–299 (2020)
18. Khanna, P., Rebeiro, C., Hazra, A.: XFC: A framework for exploitable fault characterization in block ciphers. In: Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017. pp. 8:1–8:6. ACM (2017)
19. Krcek, M., Ordas, T., Fronte, D., Picek, S.: The more you know: Improving laser fault injection with prior knowledge. In: Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2022, Virtual Event / Italy, September 16, 2022. pp. 18–29. IEEE (2022)
20. Lu, Y.: Attacking hardware AES with DFA. CoRR **abs/1902.08693** (2019)
21. 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. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(4), 56–87 (2022)
22. Piscitelli, R., Bhasin, S., Regazzoni, F.: Fault attacks, injection techniques and tools for simulation. In: 10th International Conference on Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2015, Napoli, Italy, April 21-23, 2015. pp. 1–6. IEEE (2015)
23. Porpodas, V.: ZOFI: zero-overhead fault injection tool for fast transient fault coverage analysis. CoRR **abs/1906.09390** (2019)

24. Richter-Brockmann, J., Shahmirzadi, A.R., Sasdrich, P., Moradi, A., Güneysu, T.: FIVER - robust verification of countermeasures against fault injections. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 447–473 (2021)
25. Riscure: Riscure/fisim: An open-source deterministic fault attack simulator prototype, https://github.com/Riscure/FiSim
26. Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: SAFARI: automatic synthesis of fault-attack resistant block cipher implementations. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(4), 752–765 (2020)
27. Saha, S., Alam, M., Bag, A., Mukhopadhyay, D., Dasgupta, P.: Leakage assessment in fault attacks: A deep learning perspective. IACR Cryptol. ePrint Arch. p. 306 (2020)
28. Saha, S., Jap, D., Patranabis, S., Mukhopadhyay, D., Bhasin, S., Dasgupta, P.: Automatic characterization of exploitable faults: A machine learning approach. IEEE Trans. Inf. Forensics Secur. **14**(4), 954–968 (2019)
29. Saha, S., Kumar, S.N., Patranabis, S., Mukhopadhyay, D., Dasgupta, P.: ALAFA: automatic leakage assessment for fault attack countermeasures. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019. p. 136. ACM (2019)
30. Saha, S., Kumar, U., Mukhopadhyay, D., Dasgupta, P.: An automated framework for exploitable fault identification in block ciphers - A data mining approach. In: Kühne, U., Danger, J., Guilley, S. (eds.) PROOFS 2017, 6th International Workshop on Security Proofs for Embedded Systems, Taipei, Taiwan, September 29th, 2017. EPiC Series in Computing, vol. 49, pp. 50–67. EasyChair (2017)
31. Saha, S., Mukhopadhyay, D., Dasgupta, P.: Expfault: An automated framework for exploitable fault characterization in block ciphers. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(2), 242–276 (2018)
32. 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 2013, New York City, NY, USA, October 2-4, 2013. pp. 255–260. IEEE Computer Society (2013)
33. Srivastava, M., SLPSK, P., Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: SOLOMON: an automated framework for detecting fault attack vulnerabilities in hardware. In: 2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020. pp. 310–313. IEEE (2020)
34. Timmers, N., Mune, C.: Escalating privileges in linux using voltage fault injection. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017. pp. 1–8. IEEE Computer Society (2017)
35. Timmers, N., Spruyt, A., Witteman, M.: Controlling PC on ARM using fault injection. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016. pp. 25–35. IEEE Computer Society (2016)
36. Wiersma, N., Pareja, R.: Safety != security: On the resilience of ASIL-D certified microcontrollers against fault injection attacks. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017. pp. 9–16. IEEE Computer Society (2017)
37. Wouters, L., den Herrewegen, J.V., Garcia, F.D., Oswald, D.F., Gierlichs, B., Preneel, B.: Dismantling dst80-based immobiliser systems. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(2), 99–127 (2020)