2022 CPSSI 4th International Symposium on Real-Time and Embedded Systems and Technologies (RTEST), Tehran, Iran

GCFI: A High Accurate Compiler-based Fault Injection for Transient Hardware Faults

Hussien Al-haj Ahmad, Yasser Sedaghat
Dependable Distributed Embedded Systems (DDEmS) Laboratory
Computer Engineering Department
Ferdowsi University of Mashhad
Mashhad, Iran
hussin.alhajahmad@mail.um.ac.ir, y_sedaghat@um.ac.ir

***Abstract*— Recently, with increasing system complexity and advanced technology scaling, there is a severe need for accurate fault injection (FI) techniques in the reliability evaluation of safety-critical systems against transient hardware faults, like soft errors. Since compiler-based FI techniques operate at a high intermediate representation (IR) code, their accuracy is insufficient to assess the resilience of safety-critical systems against soft errors. Although binary-level FI techniques can provide high accuracy, error propagation analysis is challenging due to missing program structures. This paper proposes an accurate GCC compiler-based FI technique called (GCFI) to assess the resilience of software against soft errors. GCFI operates at the back-end of the GCC compiler and instruments the very low-level IR code through a compiler extension. GCFI only performs instrumentation once right after the completion of optimization passes, assuring one-to-one correspondence of IR code with assembly code. The effectiveness of GCFI is evaluated by employing it to conduct many FI experiments on different benchmarks compiled for x86 and ARM architectures. We compare the results with high-level and binary-level software FI techniques to demonstrate the accuracy of GCFI. The results show that GCFI can assess the resilience of programs against soft errors with high accuracy similar to binary-level FI.**

Keywords—compiler-based fault injection, transient hardware faults, fault tolerance, assessing resilience, compiler extension.

I. INTRODUCTION

The advancements in processor technology have permitted significant increases in the transistor budgets, which have become smaller with low threshold voltages [1, 2]. As a result, highly competent and efficient processors are designed to be employed in different applications domains, such as high-performance computing and embedded systems. Regarding the latter, the popularity of embedded systems has increased dramatically, making these systems be incorporated into various applications, from information systems and heavy industries to smart cities, distributed systems, and safety-critical embedded. Since embedded systems moved to safety-critical applications, their reliability has become a dominant concern [2], as the technology advancements have negatively affected the ability of processors to tolerate faults and increased their vulnerability against transient hardware faults, e.g., soft errors [3, 4]. The occurrence of soft errors can disturb the system's execution and cause severe financial, human, or environmental disasters [2, 5, 6]. Therefore, employing fault tolerance techniques is mandatory to achieve a higher fault coverage and improve the safety-critical system's reliability [7, 8].

Quantifying the characteristics of the program's resilience to soft errors is mandatory before applying fault tolerance techniques [1, 7, 9-11]. Moreover, the target program should be evaluated to identify high-vulnerable parts against soft errors to protect them cost-effectively [6, 10, 12-14]. Thus, applying vulnerability assessment techniques such as *fault injection* (FI) is essential to develop fault-tolerant applications. Fault injection is a technique widely used to accelerate the occurrence of faults in the system under test to discover weak resources [2, 3, 15]. Moreover, fault injection facilitates designers to select the suitable technique to be applied to the target system among a wide variety of fault tolerance techniques.

Efforts to improve the fault tolerance of safety-critical systems date back decades [7]. Since safety-critical applications require reliability evaluation at a high level of accuracy, the evaluation process that employs analytical models is considered inappropriate due to overestimations and thus imprecise results [5, 11]. Therefore, fault injection has been widely employed as the baseline evaluation approach. Numerous techniques have been developed for reliability evaluation. These techniques seek to inject real-world radiation-induced errors using hardware- and software-based techniques. Regarding the former, namely accelerated beam testing [2, 5, 15], the real device is subjected to accelerated neutrons to inject faults. Due to evaluating the system's reliability in a real-world environment, such techniques can provide high accuracy and realistic results. However, due to exposing the whole device to radiation, these techniques can lead to an overestimation of the reliability indices [2, 4, 5].

Unlike hardware-based fault injection techniques, fault injection can be conducted at the software level using Software Implemented Fault Injection (SWIFI) techniques [15, 16]. SWIFI techniques can inject transient hardware faults by emulating their effects in CPU registers or memory locations. Typically, these techniques can be conducted at different levels of code granularity: (1) the high-level software, e.g., source code or the compiler's intermediate representation (IR), and (2) the low-level software, e.g., the assembly or binary code (machine code) [16-18]. The high-level IR compiler-based fault injection facilitates mapping the fault injection results to the corresponding higher-level program structures. Furthermore, fault injection into the program code, variables, data structures, and statements can be performed effectively [18, 19]. However, details regarding the program execution state are unclear at the high-level source code and IR code [20]. Typically, challenges such as compiler optimizations, missing program states, and

978-1-6654-9910-1/22/$31.00

one-to-multiple instruction translation are the main challenges that negatively impact the accuracy of high-level software FI techniques [17, 18, 21].

Several SWIFI techniques have been attempt to overcome the limitations of the high-level IR compiler-based FI by injecting faults directly into the low-level assembly or binary code [15, 18, 20, 22]. Although the binary-level fault injection can inject soft errors at high accuracy, they suffer from poor portability as they highly depend on architecture-dependent instrumentation techniques [21]. Moreover, different high-level program structures are not represented at the binary-level code, making it is difficult to correlate the obtained fault injection results with higher-level corresponding program structures.

In this paper, we propose a **G**CC **C**ompiler-based **F**ault **I**njection (GCFI) tool, which allows faults to be injected into the very low-level IR code of GNU Compiler Collection (GCC) [23] right before emitting the assembly code. Unlike the compiler-based FI techniques that operate at high-level IR code, GCFI injected faults into the very low-level GCC IR code, namely Register Transfer Language (RTL), which is very close to the assembly code. The proposed technique operates at the GCC back-end, and hence, it can perform fault injection as accurately as the binary-level fault injection. In contrast to the binary-level FI, the proposed technique can correlate the fault injection results with the corresponding high-level program structures. Therefore, GCFI facilitates the soft-error propagation analysis as the program structure is available in more detail at the back-end of the GCC compiler. One more benefit of GCFI is its high portability which facilitates performing fault injection across different architectures, various programming languages, and a wide range of embedded systems.

We evaluate the effectiveness of the proposed technique by conducting many fault injection experiments on nine diverse standard benchmark programs that cover different application domains. Moreover, we employ chi-square tests with a significance level of 0.05 to evaluate the accuracy of the proposed technique against both IR compiler-based and binary-level FI techniques. Based on the obtained results, GCFI can perform fault injection at a high level of accuracy similar to that provided by binary-level FI techniques.

The remainder of this paper is organized as follows. Section II surveys related work on software-based fault injection techniques. Section IV demonstrates the fault model adopted in this paper and gives a general background about the GCC compiling pipeline process. Section V describes the implementation and workflow of the proposed technique. A detailed evaluation is presented in Section VI. Finally, conclusions are drawn in Section VII.

II. RELATED WORK

Numerous fault injection techniques have been proposed for assessing the fault tolerance of systems against soft errors. They range from hardware-based FI techniques using physical disturbances, e.g., accelerated radiation beams, to emulate fault effects through pure software techniques. Regarding the latter, fault injection can be performed at different levels of code abstraction, e.g., at high-level software or the assembly or machine code levels.

LLFI [16], KULFI [24], and EDFI [25] inject transient hardware faults into the high-level IR code of LLVM compiler infrastructure. As the LLFI is publically available, it has been widely used in reliability evaluation studies. LLFI classifies the IR instructions into different types and injects fault, at compile-time, by instrumenting the high-level IR code. Such fault injection techniques are cross-architecture as they rely on IR code, which can be moved to different architectures. However, some information describing the program's execution state is missing at high-level and IR codes, affecting their accuracy. Compiler optimization and one-to-multiple instruction translation are other challenges that high-level IR code fault injection encounters. REFINE [21] is a compiler-based fault injection technique similar to LLFI but it performs fault injection by instrumenting the LLVM IR code at the back-end making its accuracy is better than LLFI.

Low-level FI techniques inject faults into assembly or binary code. Typically, the fault injection takes place after gaining control from the target program by employing an appropriate interrupt-based mechanism, e.g., software-trap, time-out, exception interruption, etc., or using binary instrumentation tools. Once the program's control is acquired, it is possible to corrupt the content of the CPU registers and the in-memory image of the target program. ZOFI [22] and Faultprog [19] are binary level techniques that exploit time-out and exception trigger techniques for fault injection. LDSFI [26] leverages the GNU debugger (GDB) to inject faults into binary code at run time. BIFIT [20] and PINFI [18] are examples of instrumentation-based FI techniques that rely on dynamic binary instrumentation tools, e.g., Intel Pin [27], to perform fault injection into binary code. Authors in [17] have proposed PINFI-V2 and PINFI-V3 as improved versions of PINFI.

Several studies have taken advantage of microarchitecture simulation and proposed simulation-based fault injection tools. GeFIN [28], GemV [29], and Gem5-Fim [30] are fault injection techniques that exploit the Gem5 [31], a full-system cycle-accurate simulator, for early pre-silicon vulnerability assessments. They attempt to estimate the system vulnerability through fault injection into the microarchitecture-level models of processors.

A comprehensive survey about fault injection techniques can be found in the respective survey [15].

III. MOTIVATION

The considerable overheads imposed by conventional software-based fault tolerance techniques have prompted the researchers to propose cost-effective techniques to satisfy the growing strict system requirements, such as performance and power consumption. Several studies have highlighted Selective Fault-Tolerance (SFT) as an innovative approach to reduce the overheads by considering only the high vulnerable instructions of the target program [3, 6, 9, 14]. In this context, fault injection techniques should accurately identify the most vulnerable instructions to be protected. Any defects in the fault injection technique, such as those that arise in the high-level compile-based FI techniques [16, 20, 21], could distort the results and lead to under- or over-protection [32].

Consequently, the proposed technique attempts to perform a compiler-based fault injection by instrumenting the very low-level RTL code at the back-end of the GCC compiler. Our choice of GCC is motivated by the following reasons. The GNU Compiler Collection (GCC) has been widely adopted in embedded system development. Because the GCC compiler comprises many optimization parameters, improving the assembly code can be performed effectively. This includes improving memory usage, reducing the execution time, and reducing the code size to best satisfy the system's requirements. Moreover, GCC considers sophisticated features in modern processors, e.g., out-of-order, load/store architecture, and available registers, to improve the executable code. At the back-end of the GCC compiler, the RTL code is lowered and optimized to emit the assembly code. The RTL IR code describes what each code's instruction does at a fine-grained level of detail. Therefore, instrumenting the very low-level GCC RTL code can provide accurate fault injection similar to that provided by the binary-level fault injection due to the negligible semantical gap between low-level IR at GCC back-end and assembly code.

IV. The Applied Fault model and GCC overview

In this section, we illustrate the fault model employed in this study. Moreover, the assumptions with respect to the target system are introduced. Next, we give a brief overview of the GCC compiler.

A. Fault model

It is essential to define a realistic and accurate fault model, considering the environment and application under test before conducting fault injection campaigns. A fault model should be defined in more generic terms considering the environment and the target program under test. Moreover, the fault model should be as close to real-world faults as possible to precisely imitate the real effect of faults to obtain sound results [1, 2, 5].

In this paper, the fault model we assume is a single bit-flip, as it has been widely adopted in many previous studies [2-4, 16, 22]. Moreover, authors in [1] have discussed the validity of using the single bit-flip fault model rather than the multiple bit-flip model to emulate transient hardware faults. They have proved that the single-bit flip fault model is accurate enough to be used. We model a bit-flip fault by XORing the running value at a bit location of a randomly selected architectural register with a 1. For a 32-bit architectural register, a 32-bit mask (bit flips ranging from 1 to 32) value can be used for achieving random bit-flip by XORing the selected register with the adopted mask. For example, one might flip the 3rd bit of the destination register of an instruction. We model a hardware fault as a bit-flip to be injected in architectural registers (logical registers) since the program-level logical register vulnerability factor (PVF) correlates with the Architectural Vulnerability Factor (AVF) of the physical registers as proven in previous studies [13, 33]. We assume that memory is protected by an error correction code (ECC) or parity bits. We consider a uniform distribution of fault injection campaigns. Related work has made similar assumptions [1, 18, 22].

B. Overview of GCC

The GNU Compiler Collection (GCC) is a cross-architecture, optimizing compiler widely used for different programming languages and operating systems running on various architectures. As Fig. 1 shows, the compilation pipeline of GCC comprises three steps, namely front-end, middle-end, and back-end. These steps are responsible for transforming the input into various IR codes, i.e., GENERIC, GIMPLE, and Register Transfer Language (RTL), toward producing assembly code. GENERIC IR code represents an independent programming language interface between the front-end and the subsequence steps of the compiler. GIMPLE and RTL are used during the compiling process for code optimization. Once the input is completely transformed, the GCC can emit the architecture-dependent assembly code.

In practice, a significant fraction of injected faults is detected and masked by the operating system or the underlying hardware [7]. In this context, to gain confident results, the effect of the injected faults (i.e., active/inactive faults) should be considered because the high number of active faults is likely to make results more accurate. Unlike most fault injection techniques, which work at the high-level IR code, our proposed technique operates at the very low-level GCC RTL code that is highly close to the assembly code. Therefore, GCFI can accurately analyze the program's resilience against faults.

V. GCFI: workflow and implemetation

GCFI is a compiler-based fault injection technique in which it injects faults into different instructions and operands at compile time. It leverages the GCC-plugin API provided by the GCC compiler since version 4.5 [23] in order to extend the compiler to perform fault injection. A GCC-plugin is a loadable shared object file that can be used during compiling to extend the compiler functionalities and perform custom optimizations. GCFI operates at the very low-level GCC compiler's RTL code. The RTL code is a very low-level IR code that can be seen as a generic assembly code and can be moved across different architectures. Once the target architecture is known, the GCC compiler applies the target machine description file, an architecture-dependent file describing the machine where the compiled code will run, to the RTL representation code to emit the assembly code dedicated to the target architecture [23].

The following subsection illustrates where the proposed technique adds the RTL pass in the back-end to perform instrumentation for fault injection.

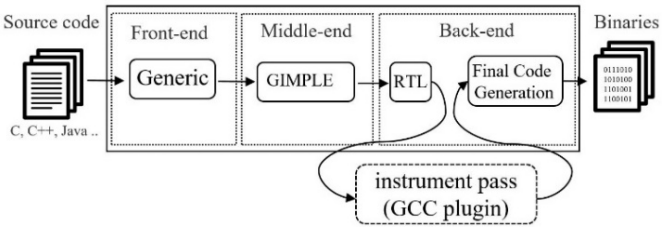


Fig. 1. Compilation pipeline of GCC compiler and the proposed plugin

A. Compiler extension

As Fig. 1 shows, GCFI inserts an RTL instrument pass by defining a GCC plugin as a compiler extension at the back-end of the GCC compiler. A compiler pass is a transformation code invoked by the GCC compiler during the compilation process to optimize and transform compilation units, e.g., functions. The final assembly code is emitted when the input unit travels through all optimization passes. The RTL pass added by the GCFI technique is responsible for instrumenting the RTL representation code right before the GCC compiler generates the assembly code. As the RTL code in the back-end accurately represents the low-level machine instructions, GCFI can precisely perform instrumentation. GCFI extends the GCC compiler by adding an RTL pass, making it a successor pass to the *pass_free_cfg*, an RTL pass executed once before emitting the assembly code. To be more specific, GCFI adds the RTL pass right before emitting assembly code and after ensuring the completion of all optimization passes. Consequently, none of the fault injection function calls (Fig. 2) added by GCFI will be affected or modified. Therefore, GCFI ensures realistic and accurate fault injection. The above-mentioned key points justify the selection of the back-end as a location for adding the RTL pass as a plugin for the GCFI technique.

B. GCFI: instrumentation mechanisim

GCFI instruments the input code, at the RTL level, by adding *function-call instruction* at determined locations, namely fault injection sites (Fig. 2). The fault injection function is responsible for injecting fault, i.e., bit-flip fault, into a specific operand of a randomly selected instruction. Fig. 2 illustrates how GCFI instruments the GCC's RTL code. In Fig. 2.a (right), we have an exclusive bitwise operation written in RTL code, which means "XORing the content of register r3 with the value of 144 and storing the result in register r3". GCFI instruments this RTL code by adding a call function instruction just before this RTL code. In the fault injection function, Fig. 2.a (left), we inject a single bit-flip fault by XORing the content of register r3 with a mask with a value of 128 (the most significant bit). Fig. 2.b shows the corresponding assembly code (for ARM architecture). At run time, before the instrumented instruction (*eor r3, r3, #144*) is executed, the program execution will transfer to the fault injection function where the bit-flip fault is explicitly injected.

GCFI instruments the program being compiled once. Thus there is no need to recompile (and re-instrument) the program version across the experiments. In other words, GCFI runs the same instrumented executable file in all the fault injection campaigns. At each fault injection experiment, a different instrumented fault injection site is selected randomly, based on the information provided by the user, to trigger the fault injection.

C. GCFI workflow

GCFI consists of three consecutive steps. As Fig. 3 illustrates, in ***Step 1***, GCFI takes multiple arguments as inputs to perform code instrumentation at compile-time. GCFI instruments the source code by defining and linking an RTL pass to the GCC back-end passes. Based on the profiling information of the program being compiled, the RTL pass determines the

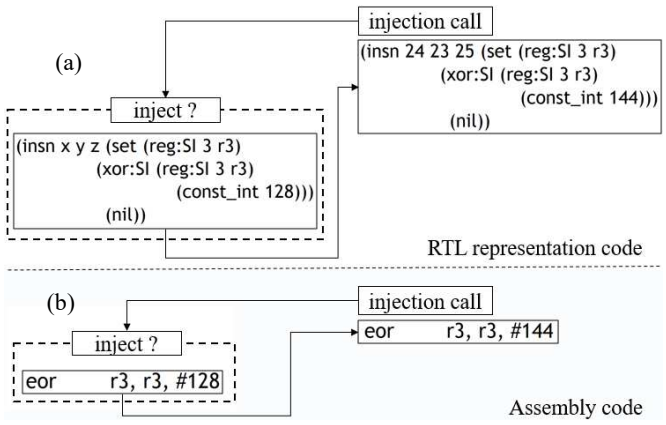


Fig. 2. Fault injection by instrumentation

appropriate instructions and marks them as fault injection sites. The profiling information is required to determine the sites within the code where the program is more likely to visit during execution. Therefore, GCFI instruments only the instructions that will be executed during the run time, reducing the size of the instrumented code. The RTL pass is implemented as a shared library plugin. GCFI uses this plugin at compile-time to instrument the code whenever it detects a user-defined fault injection site.

In ***Step 2***, GCFI performs fault injection by running the instrumented executable file. While GCFI instruments all the fault injection sites in the target program once, multiple faults may be injected because of frequently triggering the fault injection function. Recall that the candidate fault injection sites are determined by offline profiling. GCFI randomly selects one instruction, as an injection site, to trigger the fault injection function only for it, ensuring one fault injection for every execution.

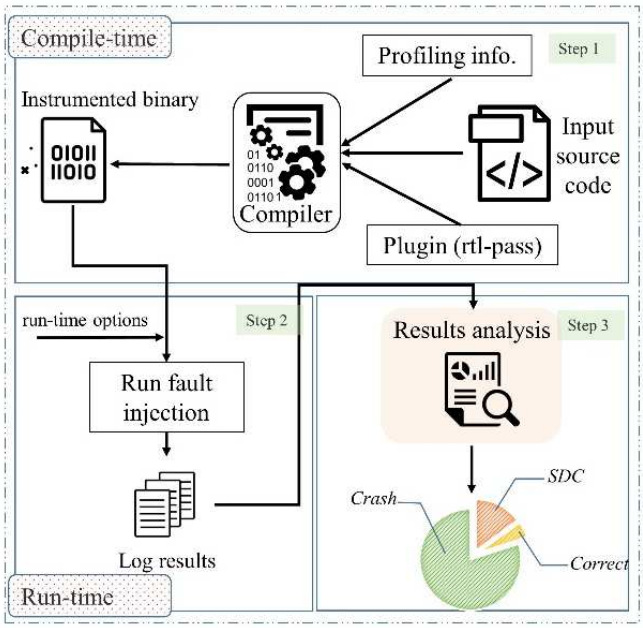


Fig. 3. Overall GCFI workflow

When the fault injection campaign is completed, GCFI arranges the produced injection results to be analyzed in ***Step 3***. Typically, a bit-flip fault in a CPU architectural register can affect the program execution and lead to different outcomes. GCFI follows previous studies [5, 16, 18, 26] in which the fault-induced effects were classified into three categories, namely Correct, Silent Data Corruption (SDC), and Crash. If the program in the presence of a fault produces golden results, fault-free execution results, the outcome is classified as Correct. If the results differ from the golden results, the outcome is classified as SDC. In both above cases, the program execution is terminated normally. If the program is abnormally terminated due to an exception, e.g., segmentation fault, the outcome is classified as a Crash.

Once the fault injection experiments are completed, GCFI generates two files for each experiment. These files are an output file that stores the result of the program under test, namely Fault Injection Output (FIO) and a log file. GCFI employs a Python script to iterate over the log and FIO files and examines their contents. The script files consider the following parameters: (1) the content of the log file, (2) the content and size of the FIO, (3) the content of the golden output. Fig. 4 illustrates the algorithm employed for classifying the effects of injected faults. When the FIO file is identical with the golden output and the corresponding log file does not indicate an exception, the fault effect category is Correct. When no exception is logged, but the output files (golden and FIO) differ in content, the fault effect is classified as SDC. If any exception is logged, the fault effect is classified as Crash.

Algorithm 1: Fault Effect Classification

```
Input: Logs, FIOs, Golden_result
Output: Fault Injection Results
/* Initialization
  Correct_Count ← 0
  SDC_Count ← 0
  Crash_Count ← 0

  /* classify fault injection outcomes as:  Correct, SDC, Crash
  for Log_file in Logs and FIO_file in FIOs do
    if ( FIO_file_size==0 OR
     Exception(Log_file)==True) then
      /* Crash is detected
      Crash_Count ← Crash_Count + 1
    end
    else if ( FIO_file_content==Golden_result AND
     Exception(Log_file)==False) then
      /* Correct is detected
      Correct_Count ← Correct_Count + 1
    end
    else if ( FIO_file_content!=Golden_result AND
     Exception(Log_file)==False) then
      /* SDC is detected
      SDC_Count ← SDC_Count + 1
    end
  end
```

Fig. 4. Pseudocode describing fault effect classification algorithm

VI. Experiments and Results

We evaluate the accuracy of GCFI in injecting faults with respect to compiler-based and binary-based fault injection techniques. As LLFI is a publicly available open-source tool [16], it has been used in several studies [1], [24], [25]. Moreover, these studies have shown that LLFI is suitable for studying different errors, particularly SDC-causing errors. Regarding binary-level fault injection, PINFI [18] is a binary-level fault injection technique that can be used for comparison. PINFI utilizes the Intel Pin [27], a dynamic binary instrumentation tool, to perform fault injection. PINFI instruments a source code that is compiled by the LLVM compiler. However, PINFI requires some modifications to render it compatible with the recent version of the Pin instrumentation tool. These modifications have led to contradictory results in different studies, such as [21] and [17]. On the other hand, LDSFI [26] represents a high-accuracy state-of-the-art binary-level fault injection technique. LDSFI injects faults into the binary code of software applications at runtime. It exploits the GNU Debugger (GDB) and injects faults through the breakpoint interruption mechanism. Moreover, LDSFI follows a uniformly distributed fault model, similar to the fault model we adopt in this study, and it relies on the GNU project, like GCFI. Therefore we adopt the LDSFI technique as an accurate baseline for comparison.

A. Benchmark programs and fault injection sites

GCFI can inject faults into different instructions categories, i.e., different fault injection sites. The instruction categories we have selected to inject faults are shown in Table I. Subsequently, one can conduct fault injection campaigns to a specific category of instructions to examine its vulnerability against soft errors. We have utilized nine diverse programs from MiBench Benchmark Suite version 1.0 [34] to evaluate the effectiveness and accuracy of the GCFI technique. The characteristics of these programs are shown in Table II. The Mibench suite includes 35 programs that fall into six categories: automotive, consumer, networking, security, office, and telecommunications. We have selected the benchmarks from different application domains to represent a wide range of scientific applications with different characteristics in terms of code size, dynamic executed instructions number, inputs, source code implementation, functionality, etc. Considering both small and large inputs provided by the benchmark suite for programs, the execution time required by MiBench programs is short, making them very convenient for fault injection experiments that need to be conducted extensively. For x86 binaries, GCFI compiles the programs with the GCC compiler, whereas the ARM binaries are built by the GCC cross-compiler to be executed on a simulated ARM system.

B. Experimental setup

As an experimental setup, we implement a prototype of GCFI that offers a GCC plugin to instrument programs and build executable files for both x86 and ARM architectures. This is possible because GCFI instruments the architecture-independent RTL representation code at compile-time. Then, the employed compiler builds an executable file using the machine description file for a specific architecture. The x86 binaries are executed on an Intel Core i7 processor, whereas the

TABLE I. FAULT INJECTION INSTRUCTION CATEGORIES

Instruction Category	Description
Arithmetic instructions	standard arithmetic operations
Load/Store instructions	data Transfer instructions
Branch instructions	control flow instructions
Where to inject?	
All possible instruction operands: (1) immediate value, (2) data registers, and (3) control registers	

TABLE II. SUMMARIZES THE ADOPTED BENCHMARK PROGRAMS AND THEIR CHARACTERISTICS

Category	Benchmark	LoC	Benchmark program description & Input
Telecomm	CRC32	1783	Calculating 32-bit Cyclic Redundancy check on 1.4MB input.pcm file
	FFT	2091	Performing the Fast Fourier Transform on a floating point data
Security	AES	4669	Encrypting an input file (small_input.asc 812 KB)
Automotive	basicmath	1985	Performing some mathematical calculation on a set of constants
	bitcount	2129	Counting the number of bits for a given array of integers
	qsort	1642	Sorting data using the quick-sort algorithm (small_input.dat ~53.4 KB)
	susan	1800	Smooths a black & white image of a rectangle
Network	dijkstra	2049	Finding the shortest paths for 2D 100x100 matrix (input.dat ~30 KB)
Office	stringsearch	2195	Searching a word in phrases

ARM binaries are executed on a bare-metal ARM architecture simulated using the cycle-accurate Gem5 simulator [31]. We instruct GCFI, LLFI, and LDSFI to instrument *load/store, branch, and arithmetic* instructions as fault injection sites. Overall, we inject 144000 faults (4 instruction types × 9 benchmarks × 4 ISAs (3 x86 + 1 ARM) × 1000 injections = 144000 injections). According to [35], our experiments correspond to a 1% error margin with a 99% confidence level.

C. Evaluation results

The proposed technique classifies the possible outcomes of fault injection experiments as follows: Correct, SDC, and Crash. Fig. 5 shows a graphical overview of the fault injection results obtained by GCFI (x86 and ARM), LLFI, and LSDFI for each benchmark and different instruction categories.

Considering all charts in Fig. 5, for the used benchmark programs (the y-axis), the x-axis represents the rate of fault injection outcomes, i.e., Correct, Crash, and SDC. As Fig. 5.a to Fig. 5.d show, the fault injection outcomes obtained by the employed techniques differ across different benchmarks and instruction categories, as fault injection into different instruction types can impact the target program in different ways [1, 9, 16]. Moreover, the fault injection results obtained by GCFI are slightly different from those obtained by the accurate binary-level fault injection. In other words, the proposed technique can perform fault injection at high accuracy.

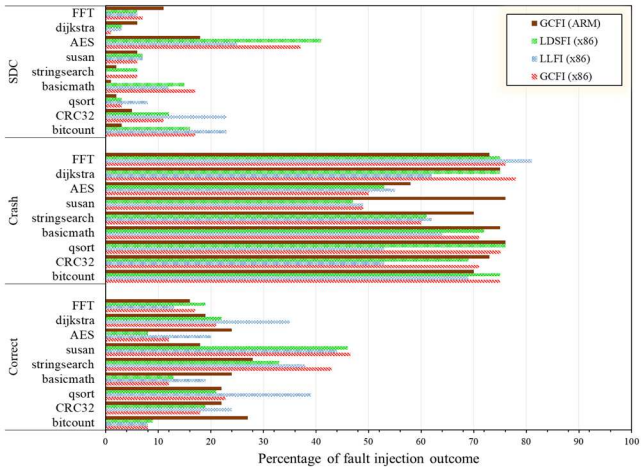
We further confirm this finding by performing a statistical analysis to evaluate the accuracy of GCFI across different employed benchmark programs and instruction categories. To evaluate the accuracy of GCFI with respect to both high-level IR software fault injection and binary-level fault injection, we employ chi-squared tests with a significance level of 0.05. To this end, we group the fault injection outcomes obtained by each technique for each benchmark program to calculate the frequencies of fault injection outcomes, as required by the chi-square tests. For example, considering fault injection performed by GCFI into the four instruction categories of bitcount benchmark program (Fig. 5.a to Fig. 5.d), the frequencies of fault injection outcomes are 776, 2592, and 632 for Correct, Crash, and SDC, respectively. For each benchmark program and pair of techniques, we calculate the contingency table of the frequencies. Since we employ nine benchmark programs, we have 18 contingency tables (nine tables for each pair of techniques).

For each benchmark program and pair of techniques, we employ the chi-square tests to compare the significance of the difference between fault injection outcomes. The chi-square test enables us to test whether there is a statistically significant difference between the expected frequencies and the observed frequencies in different categories, i.e., fault injection outcomes. To this end, we adopt a null hypothesis H0 that there is no statistically significant difference in fault injection outcome frequencies between a pair of techniques. The alternative hypothesis H1 is the opposite of the null hypothesis. Therefore, if the p-value in the chi-square tests is lower than 0.05 (the adopted significance level), we reject the null hypothesis H0 and accept the alternative one, which means there is a statistically significant difference in fault injection outcome frequencies between a pair of techniques.

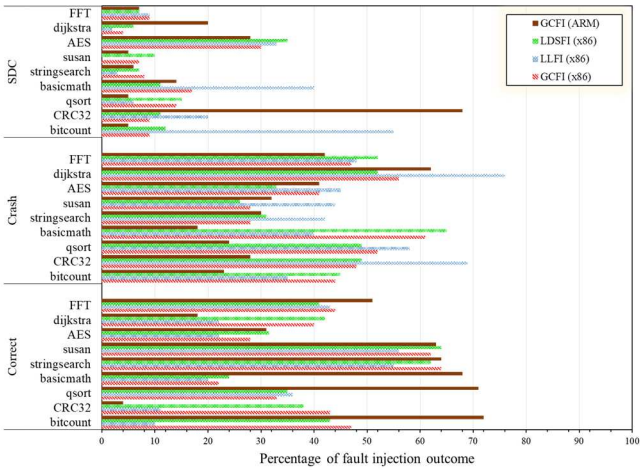
Since the binary-level fault injection is the most accurate approach, the fault injection outcomes obtained by the LDSFI technique are considered the baseline for comparison. With a confidence level of 95% and a standard significance level equal to 0.05., Table III shows the p-values of the chi-square tests. Considering fault injection performed by GCFI and LLFI into different benchmark programs, the p-values in the chi-squared tests for GCFI versus LLFI are below 0.05, indicating that we reject the null hypothesis.

TABLE III. P-VALUES OF CHI-SQUARE TESTS FOR EACH PAIR OF TECHNIQUES AND EACH BENCHMARK PROGRAMS (P-VALUE > 0.05 INDICATES THAT THE ADOPTED NULL HYPOTHESIS CANNOT BE REJECTED)

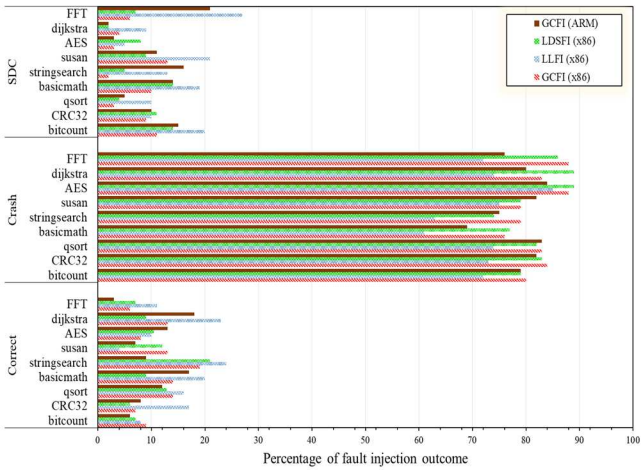
Benchmark	GCFI vs. LLFI	GCFI vs. LDSFI
bitcount	0.000	0.297
CRC32	0.000	0.052
qsort	0.000	0.254
basicmath	0.000	0.226
stringsearch	0.006	0.288
susan	0.000	0.301
AES	0.000	0.062
dijkstra	0.000	0.103
FFT	0.000	0.848



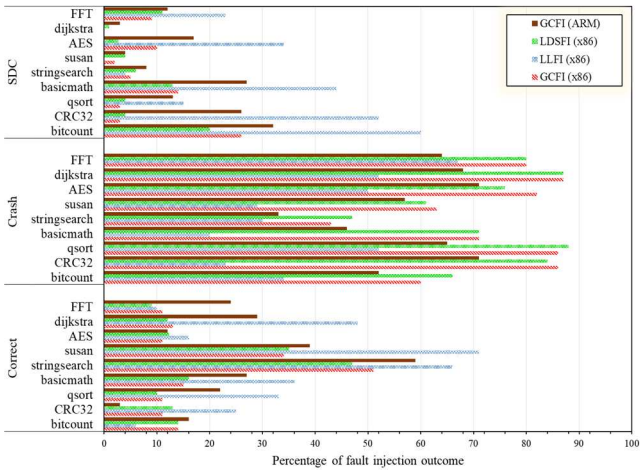
(a) fault injection outcomes into load instructions



(b) fault injection outcomes into store instructions



(c) fault injection outcomes into branch instructions



(d) fault injection outcomes into arithmetic instructions

Fig. 5. Percentage of fault injection outcomes (Correct, SDC, and Crash) for GCFI, LLFI, and LDSFI.

As a result, there is a statistically significant difference between fault injection outcomes made by GCFI and LLFI. By contrast, GCFI is not significantly different from the binary-level LDSFI technique. Based on the p-values presented in Table III, we cannot reject the null hypothesis because the p-values are higher than 0.05 (for all benchmark programs). In other words, the results obtained by GCFI and LDSFI are not statistically significantly different from each other. Therefore, the proposed technique can perform fault injection at the GCC RTL code with high accuracy similar to that provided by binary-level fault injection techniques. Moreover, the results obtained by GCFI are more accurate than those obtained by LLFI, as the GCFI's outcomes are not significantly different from those of corresponding binary-level LDSFI's outcomes.

Since the LLFI instruments the LLVM high-level IR code, it does not assure one-to-one instruction translation between the high-level IR code and the assembly code. The authors of LLFI have indicated that the gap between the high-level IR code and assembly code represents the main reason for the inaccuracy of LLFI [16, 18]. As GCFI targets RTL, which is closer than the high-level IR code to the underlying hardware on which the binaries are executed, a significant fraction of injected faults will not be masked by the system layer stack, leading to a high rate of faults activation. Our proposed technique works at the very low-level GCC compiler's RTL level. The plugin provided by GCFI executes after completing *pass_free_cfg* RTL pass, which means there are no further significant modifications before emitting the assembly code. Therefore, GCFI works at the ideal point in the compilation process to instrument the code and assure one-to-one correspondence of RTL representation code with the assembly code. It should be noted that we compare GCFI's results with those obtained by LLFI and LDSFI only for x86 binaries. We present results for ARM binaries (Fig. 5) to demonstrate that GCFI is architecture-independent and can be ported to other architectures.

VII. CONCLUSION

Assessing the resilience of safety-critical systems against soft errors is essential to reveal any defects that may lead to severe consequences, e.g., system failures. In this paper, we have proposed a compiler-based technique for fault injection into the IR code of the GCC compiler. The GCFI technique operates at a lower level of abstraction very close to assembly code, enabling highly accurate fault injection with the ability to

correlate the fault injection results with the corresponding high-level program structures. The GCFI has been validated by comparing its accuracy with state-of-the-art high-level compiler-based/binary-level fault injection techniques and tested through a large number of injection campaigns. The results show that GCFI can perform highly accurate fault injection similar to binary-level fault injection. Therefore, resilience studies can benefit from GCFI as it best satisfies the required level of accuracy. Moreover, GCFI can be customized for different architectures with negligible effort related to compiler configuration.

REFERENCES

[1] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors in Programs," *IEEE Transactions on Dependable and Secure Computing,* 2020.

[2] P. Bodmann, G. Papadimitriou, R. L. R. Junior, D. Gizopoulos, and P. Rech, "Soft Error Effects on Arm Microprocessors: Early Estimations vs. Chip Measurements," IEEE Transactions on Computers, 2021.

[3] Z. Li, H. Menon, K. Mohror, P.-T. Bremer, Y. Livant, and V. Pascucci, "Understanding a program's resiliency through error propagation," in Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2021, pp. 362-373.

[4] P. M. Aviles, A. Lindoso, J. A. Belloch, and L. Entrena, "Evaluating reliability through soft error triggered exceptions at ARM Cortex-A9 microprocessor," Microelectronics Reliability, pp. 114323, 2021.

[5] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech, "Demystifying soft error assessment strategies on ARM CPUs: Microarchitectural fault injection vs. neutron beam experiments," in 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019, pp. 26-38.

[6] A. Mokhtarpour, A. M. H. Monazzah, and H. Farbeh, "PB-IFMC: A Selective Soft Error Protection Method Based on Instruction Fault Masking Capability," in 25th International Computer Conference, Computer Society of Iran (CSICC), 2020, pp. 1-9.

[7] H. So, M. Didehban, Y. Ko, R. Jeyapaul, J. Kim, Y. Kim, et al., "Revisiting Symptom-Based Fault Tolerant Techniques against Soft Errors," Electronics, vol. 10, pp. 3028, 2021.

[8] A. Tajary, H. R. Zarandi, and N. Bagherzadeh, "IRHT: An SDC detection and recovery architecture based on value locality of instruction binary codes," Microprocessors and Microsystems, vol. 77, pp. 103159, 2020.

[9] J. A. Martínez, J. A. Maestro, and P. Reviriego, "Evaluating the impact of the instruction set on microprocessor reliability to soft errors," IEEE Transactions on Device and Materials Reliability, vol. 18, pp. 70-79, 2018.

[10] G. Papadimitriou and D. Gizopoulos, "Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers," in ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021, pp. 902-915.

[11] Y. Ko, "Characterizing System-Level Masking Effects against Soft Errors," Electronics, vol. 10, pp. 2286, 2021.

[12] X. Xu and M.-L. Li, "Understanding soft error propagation using efficient vulnerability-driven fault injection," in IEEE International Conference on Dependable Systems and Networks (DSN), 2012, pp. 1-12.

[13] V. Sridharan and D. R. Kaeli, "Quantifying software vulnerability," presented at the Proceedings of the workshop on Radiation effects and fault tolerance in nanometer technologies, Ischia, Italy, 2008.

[14] S. Arslan and O. Unsal, "Efficient selective replication of critical code regions for SDC mitigation leveraging redundant multithreading," The Journal of Supercomputing, pp. 1-31, 2021.

[15] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A Systematic Review on Software Robustness Assessment," ACM Computing Surveys (CSUR), vol. 54, pp. 1-65, 2021.

[16] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: An intermediate code-level fault injection tool for hardware faults," in IEEE International Conference on Software Quality, Reliability and Security, 2015, pp. 11-16.

[17] L. Palazzi, G. Li, B. Fang, and K. Pattabiraman, "A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection," in IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), 2019, pp. 151-162.

[18] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2014, pp. 375-382.

[19] D. Cotroneo, A. Lanzaro, and R. Natella, "Faultprog: Testing the accuracy of binary-level software fault injection," IEEE Transactions on Dependable and Secure Computing, vol. 15, pp. 40-53, 2018.

[20] Y. Wang, J. Dong, S. Zhang, and D. Zuo, "B-SEFI: A Binary Level Soft Error Fault Injection Tool," in IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2019, pp. 235-241.

[21] G. Georgakoudis, I. Laguna, D. S. Nikolopoulos, and M. Schulz, "Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017, pp. 1-14.

[22] V. Porpodas, "ZOFI: Zero-Overhead Fault Injection Tool for Fast Transient Fault Coverage Analysis," arXiv preprint arXiv:1906.09390, 2019.

[23] GNU Compiler Collection Internals for GCC Version 5.2.0, Free Software Foundation, Inc., Boston, MA, USA, 2015.

[24] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in IEEE 19th Pacific Rim International Symposium on Dependable Computing, 2013, pp. 41-50.

[25] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "EDFI: A dependable fault injection tool for dependability benchmarking experiments," in IEEE 19th Pacific Rim International Symposium on Dependable Computing, 2013, pp. 31-40.

[26] H. A.-h. Ahmad, Y. Sedaghat, and M. Moradiyan, "LDSFI: a Lightweight Dynamic Software-based Fault Injection," in 9th International Conference on Computer and Knowledge Engineering (ICCKE), 2019, pp. 207-213.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, et al., "Pin: building customized program analysis tools with dynamic instrumentation," Acm sigplan notices, vol. 40, pp. 190-200, 2005.

[28] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in Workload Characterization (IISWC), IEEE International Symposium on, 2015, pp. 172-182.

[29] K. Tanikella, Y. Koy, R. Jeyapaul, K. Lee, and A. Shrivastava, "gemV: A validated toolset for the early exploration of system reliability," in IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2016, pp. 159-163.

[30] F. R. Da Rosa, R. Reis, and L. Ost, "gem5-FIM: a flexible and scalable multicore soft error assessment framework to early reliability design space explorations," in IEEE 9th Latin American Symposium on Circuits & Systems (LASCAS), 2018, pp. 1-4.

[31] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, et al., "The gem5 simulator," ACM SIGARCH Computer Architecture News, vol. 39, pp. 1-7, 2011.

[32] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2018, pp. 27-38.

[33] G. S. Rodrigues, F. Rosa, Á. B. de Oliveira, F. L. Kastensmidt, L. Ost, and R. Reis, "Analyzing the Impact of Fault-Tolerance Methods in ARM Processors Under Soft Errors Running Linux and Parallelization APIs," IEEE Transactions on Nuclear Science, vol. 64, pp. 2196-2203, 2017.

[34] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538), 2001, pp. 3-14.

[35] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in Design, Automation & Test in Europe Conference & Exhibition, 2009, pp. 502-506.