

# LDSFI: a Lightweight Dynamic Software-based Fault Injection

Hussien Al-haj Ahmad, Yasser Sedaghat, Mahin Moradiyan

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, m\_moradiyan@mail.um.ac.ir

**Abstract**— Recently, numerous safety-critical systems have employed a variety of fault tolerance techniques, which are considered an essential requirement to keep the system fault-tolerant. While the current trend in processors technology has increased their effectiveness and performance, the sensitivity of processors to soft errors has increased significantly, making their fault tolerance ability questionable. In this context, fault injection is considered as one of the most popular, rapid, and cost-effective techniques which enables the designers to assess the fault tolerance of systems under faults before their deployment. In this paper, a pure software fault injection technique called LDSFI (a Lightweight Dynamic Software-based Fault Injection) is presented and evaluated. Due to the dynamic aspect of LDSFI, faults are automatically injected into binary code at runtime. Thereby, the proposed technique does not impose any program runtime overhead since the intended source code is not required. The effectiveness of LDSFI was validated through performing exhaustive fault injection experiments using well-known benchmarks. The experiments were carried out using a Core 2 Duo processor, as an Intel x86 Dual-Core PC with 4GB RAM running Ubuntu Linux 14.04 with the GNU Compiler Collection (GCC) version 4.9. Since LDSFI relies on the GNU, it is highly portable and can be adapted for different platforms.

**Keywords**— *software-implemented fault injection; faults, fault tolerance; soft error; dynamic binary injection.*

## I. INTRODUCTION

Nowadays, processor manufacturing has improved efficaciously which has led to the emergence of more powerful, newer generations of processors in term of performance and efficiency. These advances in processors technology have permitted significant increases in the transistor budgets, which have become smaller with low threshold voltages [1]. Despite the fact that this trend has yielded performance enhancements and increased their effectiveness compared to the previous generations, it has led to increase complexity and make the processors more susceptible to external events such as highly energized particles [2-6]. Strictly speaking, the advancements in processor manufacturing have negatively affected their robustness, making them more sensitive to soft errors.

Regard to safety-critical systems where fault tolerance is considered a significant requirement, evaluating the system vulnerability to soft errors is crucial [7]. Hence, to make the system fault-tolerant, exploiting fault tolerance techniques to achieve system requirements like safety and reliability, is unavoidable. Accordingly, using the targeted fault injection technique can provide a powerful and useful solution to evaluate systems under faults [8-12]. Basically, the fault injection technique is essential needed to assess fault-tolerant systems. It is a practical approach that uses to accelerate the

occurrence of faults in the system and then monitor the system's behavior for vulnerability assessment against faults [4, 11-13]. Thanks to the fault injection techniques, the faults can be injected in a fully focused style, either architecturally in the ISA or micro-architecturally into CPU registers [7].

Prior to performing fault injection, the potential effects induced by faults should be modeled. In addition, the adopted fault model should be a realistic, suitable model able to reflect the real effect of soft errors [14, 15]. Generally, hardware faults that arise in systems can be classified into three categories depending on the length of time they persist: permanent, transient, and intermittent [12, 16, 17]. Design defects and bugs, either in hardware or software, are considered the main reasons for the permanent faults. These faults can impede the entire system functionality, causing dangerous performance degradation and energy consumption [16]. Transient and intermittent faults which are induced by environmental effects like gamma-rays and high-energy neutrons, do not result in permanent damage. However, they can lead to wrong execution of the program due to flipping single or multiple bits either in memory or CPU registers [16]. Based on previous studies, transient faults (a.k.a. soft errors) are considered as a major disturbance to safety-critical systems and can lead to data-flow error and control flow errors [7, 18, 19]. Accordingly, soft errors, e.g., the Single Event Upset (SEU) such as bit-flips, is the adopted fault model used by the proposed technique in order to emulate the occurrence of faults.

For the evaluation purpose, the fault injection techniques have been used to evaluate a plethora of techniques in many different research fields related to fault tolerance [4, 8, 12, 20]. Fault injection techniques fall into three different categories, namely: software-based techniques, hardware-based techniques, and a combination of software and hardware so-called hybrid techniques [12].

In the early 1970s, the fault injection technique was used at the hardware level. Hardware-based fault injection can be carried out either externally in pin level or internally with radiation interference [12]. Regard to internally techniques, also called radiation-testing, accelerated radiation beams are generated to strike a memory or CPU. Hardware Implemented Fault Injection (HWIFI) techniques evaluate the fault tolerance of a given system through injecting hardware faults. Also, HWIFI is the most realistic way to check the effects of a fault on the target system at the physical level and measure the failure rate [4]. HWIFI puts the target system being evaluated in a real-world evaluation harsh environment. MESSALINE [21], FIST [22], and MARS [23] are examples of HWIFI tools techniques presented in the literature. MESSALINE can inject

faults in stuck-at and complex logical. Also, it has the ability to control the length and frequency of the injected fault [21]. Using heavy ion radiation, FIST generates one-bit or multiple bits of transient faults inside random locations of the chip [22]. MARS is a fault-tolerant architecture which can be used to perform fault injection in two different fashions: 1) an electromagnetic-based fault injection, and 2) exploiting the heavy ion radiation technique provided by FIST tool [23].

Despite the variety of techniques that aim at injecting faults using different methodologies, radiation-testing remains the most accurate technique to assess the fault-tolerant systems. However, with all the benefits gained from using radiation-testing, major drawbacks still exist. It is not simple to achieve a real-world environment due to the dramatic cost involved in establishing (difficult to set up) such environments [8, 12]. In addition, radiation-testing can complicate the way to determine the main source of the error or where it occurred precisely. Additionally, while evaluating the system under faults, a part of memory should be kept out the radiation range to store a golden data, fault-free, to be used later in comparing the results of experiments to detect any mismatch. Strictly speaking, HWIFI seems to be not cost-effective due to utilizing additional special-purpose hardware components and the need to rebuilding the target system in case of failures [12, 24].

Hybrid fault injection approach provides robust and more effective techniques of fault injection through exploiting the advantages of both software-based and hardware-based techniques [8, 12, 25, 26]. The method presented in [25] can inject numerous types of faults, such as transient and permanent faults, into circuits which should be described using hardware description language. However, hybrid techniques should be carefully designed, taking several considerations into account like time and performance overheads. Also, due to the interaction between software and hardware components, performance might degrade. Additionally, since many hybrid techniques use FPGA to perform fault injection and analyze the error propagation, such techniques suffer from multiple drawbacks including the low flexibility and low adaptability, the low scalability, financial cost, and the complexities involving exchange the fault model or implement a comprehensive one [8, 24, 26].

In Software-Implemented Fault Injection (SWIFI), soft errors are injected using pure software ways without the need for any additional special-purpose hardware [12, 13, 27, 28]. Taking into consideration the different program access levels, the injection process can be performed either at the program source code level or at the binary code level. [13, 29]. The former aims at instrumenting the source code to inject faults. Hence, the accessibility of source code is required, which cannot always be available. In addition, with any changes performed on the source code, recompilation is required, which could result in a significant increase in time overhead for large programs across the experiments [29]. On the contrary, fault injection at the binary level is applicable even with no source code available. It can be performed at runtime on the binary code after it has been compiled [13, 29]. Consequently, fault injection at a binary level can be done very efficiently without runtime overhead due to eliminating the need for recompilation. The technique suggested in this paper falls

under the binary level fault injection category and uses The GNU Debugger (GDB) to inject faults at runtime. GDB is a well-known debugger that supports different programming languages and processor architectures. LDSFI exploits GDB to inject faults at runtime by interrupting the program execution periodically using software interrupts (a.k.a. "traps"). Also, LDSFI can work at binary code level without any prior knowledge about the source code. It is a portable and easy-to-use technique which enables the designers to analyze and improve the fault tolerance of safety-critical systems. Thereby, LDSFI facilitates the way to make the comparison results more confident in a cost-effective manner by performing exhaustive fault injection experiments in near real-time. As a result, the benefits gained from the proposed technique are reflected in reducing the cost and effort required to perform enormous experiments. Moreover, the designers have also focused on different error detection techniques like data-flow and control-flow to improve the system ability to tolerate faults. In this context, it is prudent to make a given technique capable of injecting faults that will result in errors either in the data flow or the control flow of the program. LDSFI has an ability to inject errors into the code, at the binary level, of the target program.

The later sections in this paper are organized as follows: the related work on SWIFI techniques is summarized in Section II, which also highlights some of the most common techniques presented in the literature. Section III discusses in greater detail of the philosophy of LDSFI and the adopted fault model. Evaluation of the proposed method is discussed in Section IV. Finally, Section V concludes this work and gives a brief about future work.

## II. RELATED WORK

Recently, SWIFI techniques has attracted designers' attention since changing or reconfiguring the hardware of the target system is not required. In addition, SWIFI can provide a suitable assessment scheme which allows designers to inject faults either into software applications or operating systems. It is important to mention that most of SWIFI techniques inject faults either statically or dynamically [13, 15]. In the following subsection, static and dynamic fault injection are discussed.

### A. Static and Dynamic Injection

Based on the characteristics of SWIFI, static and dynamic injection is possible [13, 15, 29]. Regard to the former, an additional set of instructions responsible for injecting a particular fault is inserted into the program source code. Such methodologies can operate either at source code level or binary code level using static binary instrumenting frameworks. However, static injection usually incurs additional overhead for recompiling the instrumented code [29]. What distinguishes dynamic injection from static one is that it works directly at the binary code level, which means that there is no need to recompile the modified version across the experiments. As a result, no additional overhead is required. Dynamic techniques use a triggering mechanism in order to inject faults at run-time. Commonly used triggering mechanisms are [28, 30]:

- *Time-out.* The defined faults are injected when the used timer expires and produces a time-out event, which in turn creates an interruption. This timer can be set up using hardware or software methods. No program modifications required. Time-out mechanism is suitable for injecting hardware faults, e.g., transient and intermittent.
- *Exception/trap.* The fault injection process is controlled by the occurrence of an exception or a trap. The exception/trap mechanism injects faults whenever certain conditions occur; for instance, a fault can be injected before executing a particular instruction.
- *Code insertion.* This mechanism is similar to the code modification technique, where special-purpose instructions are inserted inside the code. However, the code insertion mechanism aims at inserting instructions during the program execution to modify the original instructions.

### B. Fault Injection Accuracy

Different factors impact the results of fault injection. These factors should be taken into consideration when adopted a new technique to inject faults. These factors are as follow [15]:

- *Representativeness.* The fault model applied to target programs should be robustness and accurate to reproduce the real faults. By doing so, results with high confidence can be gained from the performed experiments.
- *Non-intrusiveness.* The fault injection campaign should not result in a significant remarkable change in the workflow of the program contrary to what is defined in the adopted fault model. Fault injection techniques that rely on source code instrumentation should take this challenge more seriously so that no excess instructions are executed; particularly for the real-time systems.
- *Repeatability.* Taking several considerations into account like the same case-studies and experimental environment set up, the fault injection technique should ensure that the gained results do not show significant run-to-run variation, two different runs of the same program with exactly same fault injection campaign should be identical.
- *Practicability.* Some fault injection techniques incur overheads in terms of cost and time. Practicability indicates the effectiveness provided by a given technique to perform fault injection at a suitable cost with reasonable time. As mentioned earlier, the fault injection campaign should take into consideration the time conditions while injecting faults; to name some: the time needed to establish and set up the experiment environment, performing the injection, gained and analysis the results. To obtain confident results, a large number of faults must be injected. Hence, it is obvious that the whole steps of fault injection should be performed as automatic as possible.
- *Portability.* Indicates the usability of the same fault injection tool across different environments for different case studies. Portability is necessary to enable researchers to make empirical comparison with different studies. In Addition, the adapted tool should be compatible with different fault models.

### C. Overview of SWIFI Techniques

For the sake of brevity, commonly used SWIFI techniques are discussed here. Ferrari [31] injects faults into the CPU,

memory, and Bus based on the traps technique. It consists of four main parts: the initializer and activator, the user information about faults to be injected, the fault-and-error injector, and the data collector and analyzer. The injected faults can be transient or permanent faults [31]. Another example of SWIFI technique is FTAPE, which injects faults into CPU registers, memory locations, and into other peripherals [32]. To inject faults into the peripherals, a special-purpose routine is executing in the driver code, which emulates the I/O errors (for example, bus error and timer error). Xception [14] uses advanced debugging features in modern processors, called hardware exception triggers, to inject faults. Time-out and exception trigger techniques are used in INERTE [33] to inject faults into the system memory. Three main modules comprise INERTE: Experiment Generator Module (EGM), Fault Injector (FI), and Analysis Tool (AT). EGM receives fault injection data as inputs and then generates the configuration files (CF). The fault injection data determine the code space where faults can be injected and the number of injected faults. Then, the fault injection is performed by FI. Finally, the fault injection results are analyzed by the AT module. LLFI [34] injects faults into the target system in two phases; initially, using LLVM compiler, the source code is converted to LLVM IR (a low-level intermediate representation) at compile time. Also, specific program points where the faults can be injected are determined by information provided from the user to the compiler. During the second phase, i.e., at runtime, two executable files are generated, fault injection and profiling files. The former is responsible for performing fault injection into the target system by utilizing the profiling file. Finally, DDSFIS [13] is a software-based technique uses GDB to perform fault injection. DDSFIS injects faults at runtime in an automated fashion without recompiling. However, primary information should be extracted by analyzing the source code statically. As a result, on the opposite of LDSFI, the source code should be available. Also, for analyzing the source code, additional software is required, which limits its portability and makes it applicable only when both the source code and Eclipse CDT are available. Moreover, since DDSFIS works at the source code level, its accuracy might be affected by compiler optimizations. All the above have pointed out the benefits gained from SWIFI techniques. However, such techniques suffer from the following shortcomings [20]:

- Since some of the internal components of CPU are inaccessible by the software system, SWIFI techniques seem to be limited only to accessible components.
- SWIFI tools that rely on the source code instrumentation, should consider the system requirements, particularly for real-time systems that need a strict response deadline.

The proposed technique in this paper seeks to satisfy the fault injection accuracy factors discussed earlier. LDSFI uses a robustness, accurate fault model to imitate the real effect of bit-flip errors. Also, LDSFI involves several steps where each step is executed automatically. As a result, the time-consumption required to perform fault injection is decreased. Moreover, the proposed technique ensures that two different runs of the same program with exactly the same fault injection campaign, using the same fault model, should be identical.

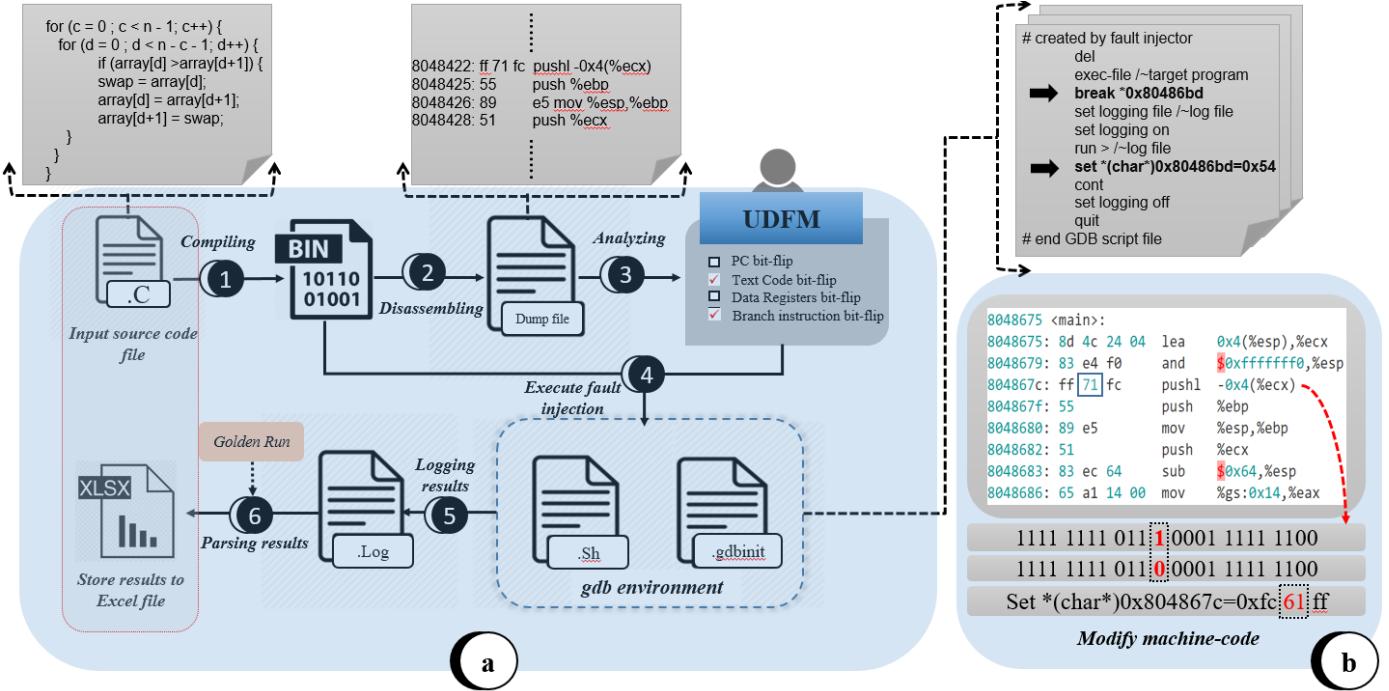


Fig 1. (a) The LDSFI fault injection steps, (b) An example of a script file sample generated automatically by LDSFI

Since LDSFI is based on the GNU debugger (GDB), it is a cross-platform technique, which can be widely used to make an empirical comparison between different studies. Therefore, LDSFI facilitates the evaluation of fault tolerance techniques against errors by performing exhaustive experiments automatically.

### III. THE PROPOSED METHOD

In this paper, a new SWIFI tool so-called LDSFI is presented. The idea behind the fault injection from the software perspective is injecting soft errors to imitate the real hardware faults induced by external disturbances, e.g., radiations.

In general, SWIFI techniques share the same fault injection steps [13, 29]. In the first step, the target program should be compiled to executable form. In the second step, regard to static fault injection techniques, faults are injected during the compilation process of the program. On the opposite, the dynamic fault injection techniques can inject faults at runtime. Also, the second step should be accompanied by a definition of a fault model that describes the faults being injected. In addition, such techniques use triggering mechanisms, e.g., exception/trap, during fault injection. In the final step, the outputs obtained from injecting faults should be analyzed to report the final results. LDSFI follows the same steps to inject faults, as later sections discuss.

#### A. LDSFI: The Supported Fault Model

Based on previous studies, the SEU is the most well-known fault model that has been adopted by several fault injection techniques [8, 35]. Taking the *w-three* [15] criteria into account (what to inject?, where to inject?, and when to inject?), the adopted fault model should take into consideration several policies like fault location and type, injection time, and the

observation scheme of fault effects. Consequently, a wrapper fault model named user-defined fault model (UDFM) based on SEU is introduced to facilitate and clarify the fault injection campaign. The UDFM is defined flexibly by the user and applied to the target program automatically. Also, it specifies the type and location where the faults, i.e., bit-flips, are going to be injected. These faults are randomly generated and injected into the target program as they defined in the UDFM.

The user builds the intended UDFM to be applied automatically to the target program. The UDFM consists of multiple fault types that will be injected at the binary code level. Since most of the studies to assess the fault tolerance of systems have been conducted experiments using the SEU fault model, e.g., bit-flips [8, 26, 29, 30, 36, 37], LDSFI aims at injecting faults in different locations (memory, address registers, data registers, and program counter) as follows:

- Injecting a bit-flip in the program counter (PC) register,
- Injecting a bit-flip in a random instruction,
- Injecting a bit-flip in random data registers,
- Injecting a bit-flip in a branch instruction,
- Replacing a branch instruction with a non-branch one,
- Removing a branch instruction.

Many techniques have been developed to protect systems against data-flow and control-flow errors. Therefore, to evaluate error detection techniques, LDSFI attempts to emulate the appearance of faults which in turn can result in control flow or data flow error. UDFM can produce such errors through corrupting either a particular branch instruction (replacing or removing a branch instruction) or the content of data registers. Regarding the *w-three* criteria discussed in this section, the faults defined in UDFM are injected automatically into the code at runtime in a random fashion.

### B. LDSFI: Implementation Details

In order to perform an accurate and effective fault injection at the binary code level, some meta-information should be derived from the target binary file like memory addresses/offsets and assembly code. LDSFI aims at injecting faults into the binary code which is compiled for the Intel x86 instruction set (ISA). Using disassembling techniques widely used in the binary world, it is possible to get down into the details of the target binary code and recover the assembly code by disassembling the binary machine code. In this context, LDSFI uses the GNU binary utility called *objdump* to perform disassembling, and generate an architecture-specific disassembled file to be analyzed. By doing so, it is possible to analyze the aforementioned file and recognize the machine instructions and their physical addresses, opcodes and operands, and registers used by the program at runtime. LDSFI uses the information extracted from the disassembled file as a basic guideline to specify locations in code space where to inject faults. Accordingly, it is possible to imitate the soft errors by flipping a single bit at a particular location, e.g., at registers, opcodes, and operands.

As illustrated in Fig. 1 (a), LDSFI involves several steps which are done automatically with a minor involvement from users in the definition of the UDFM. Also, LDSFI can operate either on the high-level source code after compilation or on the binary code level as Fig. 1 (b) illustrates. In both cases, no prior knowledge about the source code is required. Thus, LDSFI can inject faults, even with no availability of the source code. Additionally, although LDSFI is mainly focused on the Intel x86 architecture, it can be easily adapted for different x86 architectures with a minor effort.

To perform fault injection process in a fully automated fashion, GDB initialization scripts are generated based on the UDFM. The GDB script file is a user-friendly GDB initialization file comprises several commands to guide GDB to perform fault injection automatically. Once the fault injection process is completed, the corresponding results can be logged into separate log files. Finally, to obtain results from the whole fault injection process, all log files are carefully parsed, i.e., examining the content, and the results are reported. The fault injection steps, as illustrated in Fig. 1 (a), are summarized as follows:

- *Compiling*. Compiling the input file and building a binary code.
- *Disassembling*. Disassembling the binary file to extract meta-information related to the compiled file.
- *Analyzing*. Analyzing meta-information and generating scripts files based on UDFM.
- *Fault Injecting*. Performing fault injection automatically.
- *Logging*. Logging fault injection results.
- *Parsing*. Parsing logs file and generating final results.

## IV. EVALUATION OF THE PROPOSED TECHNIQUE

In order to assess the efficiency of LDSFI, an experimental environment included a Core 2 Duo processor, as an Intel x86 Dual-Core PC with 4GB RAM running Ubuntu Linux 14.04 is built. Also, similar to most of the studies presented in the literature, most of the techniques for evaluating the system

under faults, use common benchmarks such as bubble sort (BS), quicksort (QS), and matrix multiplication (MM) [14, 37-39], these benchmarks were used and submitted to fault injection in the experiments conducted in this paper. As illustrated in TABLE I, it presents the number of faults to inject and the results obtained from the experiments applied to each benchmark. These results can be classified into four different cases, namely [37]:

- OS (Operating System): The program might produce an exception due to the deviation in the program's execution induced by faults, mostly detected by the operating system.
- TO (Timeout): the injected fault affects the execution time of the program making the program consumes more time than expected, e.g., the program gets stuck in an infinite loop.
- CR (Correct Result): the injected fault has no effects on the program, and the final result is correct.
- SDC (Silent Data Corruption): the final result of program execution is not correct as the injected faults change the program behavior.

The results presented in TABLE I show that LDSFI is capable of injecting faults into the binary code and data of the running program. Since recompiling the source code is not required during fault injection, the experiments can be performed as fast as possible with no substantial runtime overhead. Strictly speaking, as LDSFI technique works at the binary code level without any source-level instrumentation, this technique does not impose any program (source code) runtime overhead. Consequently, LDSFI is an effective and suitable technique, which can be utilized for fault injection on real-time systems.

By surveying numerous SWIFI techniques, a suitable, closest techniques are compared with LDSFI, taking into consideration such factors as:

- Source code: indicates whether the source code is required to perform fault injection or not.
- Injection level: indicates whether the fault injection is performed at the binary code level, i.e., instruction level, or at the source code level, statement level.
- Fault model: refers to the fault model used in the corresponding fault injection technique.
- Portability: indicates the portability provided by the fault injection technique across different environments.
- Injection time: indicates whether the faults are injected at runtime or during program compilation time.

TABLE II depicts the comparison between LDSFI and other SWIFI techniques. As shown in TABLE II, LDSFI is an effective technique that can facilitate to designers to examine the fault tolerance of systems.

## V. CONCLUSION AND FUTURE WORK

Fault injection is quite necessary for the development process of many systems, particularly safety-critical ones. Such systems operate in harsh environments and are prone to failure by soft errors. Fault tolerance techniques are usually adopted for error detection and improve system reliability. For

TABLE I: RESULTS OF FAULT INJECTION EXPERIMENTS

#Fa	Benchmark	FLIPPING SINGLE BIT (SEU)												MODIFY CODE					
		PC <sup>b</sup>			DR <sup>c</sup>			TC <sup>d</sup>			BI <sup>e</sup>			IBI <sup>f</sup>			RBI <sup>g</sup>		
		MM	BS	QS	MM	BS	QS	MM	BS	QS	MM	BS	QS	MM	BS	QS	MM	BS	QS
500	OS	245	284	271	233	205	219	266	280	316	203	231	199	272	225	228	195	176	114
	TO	215	183	209	146	191	176	204	186	164	247	246	219	209	260	243	256	270	232
	CO	17	14	9	28	31	34	21	18	8	43	19	61	7	9	12	38	33	46
	SDC	23	19	11	93	73	71	9	16	12	7	4	21	12	6	17	11	21	17
1K	OS	478	576	637	536	335	430	378	548	599	537	463	458	510	581	461	384	428	477
	TO	459	357	293	213	444	345	497	340	302	356	491	399	412	376	456	534	472	417
	CR	26	36	23	64	87	89	114	66	35	76	27	98	22	19	39	51	44	64
	SDC	37	31	47	187	114	136	11	46	64	31	19	45	56	24	44	31	56	42

- a. No. of faults
- b. Fault injection in Program Counter
- c. Fault injection in Data general Registers
- d. Fault injection in machine code
- e. Fault injection in Branch Instruction
- f. Inserting a Branch Instruction
- g. Replacing a Branch Instruction

TABLE II: LDSFI VS. COMMON SWIFI TECHNIQUES

FACTORS	LDSFI	DDSFIS [13]	INERTE [33]	LLFI [34]
Source code	Not required	Required	Required	Required
Injection level	Instruction level	Statement Level	Statement Level	Statement Level
Fault model	Bit-flip + Code-modification	Code-modification	Bit-flip	Bit-flip
Portability	High	Medium	Medium	Medium
Injection time	Run-time	Run-time	Run-time	Run-time

examining the effectiveness and evaluating the robustness of a given system under fault, almost always fault injection techniques are exploited. In this paper, a SWIFI technique called LDSFI is presented. Since LDSFI relies on the GNU Debugger, it is a cross-platform technique, which can be easily adapted to run on different platforms. Since ARM processors are frequently used in plenty of safety-critical systems, for future work, the main goal is to expand LDSFI tool to cover the ARM architecture, in addition to the current support provided to the Intel x86 architecture.

## REFERENCES

- [1] R. C. Baumann, "Soft errors in advanced semiconductor devices-part I: the three radiation sources," *IEEE Transactions on device and materials reliability*, vol. 1, pp. 17-22, 2001.
- [2] A. Rhisheekesan, R. Jeyapaul, and A. Shrivastava, "Control Flow Checking or Not?(for Soft Errors)," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, p. 11, 2019.
- [3] V. B. Thati, J. Vankeirsbilck, N. Penneman, D. Pissoort and J. Boydens, "An Improved Data Error Detection Technique for Dependable Embedded Software," *IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, Taipei, Taiwan, 2018, pp. 213-220.
- [4] Y. S. Jeong, S. M. Lee, and S. E. Lee, "A Survey of Fault-Injection Methodologies for Soft Error Rate Modeling in Systems-on-Chips," *Bulletin of Electrical Engineering and Informatics*, vol. 5, pp. 169-177, 2016.
- [5] S. A. Keshtgar and B. B. Arasteh, "Enhancing software reliability against soft-error using minimum redundancy on critical data," *International Journal of Computer Network and Information Security*, vol. 9, p. 21, 2017.
- [6] L. Wei and M. Xu, "Enhancing Software Reliability Against Soft Error Using Critical Data Model," in *International Conference on Mobile Ad-Hoc and Sensor Networks*, 2017, pp. 365-376.
- [7] 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.
- [8] L. Entrena, M. García-Valderrama, A. Lindoso, M. Portela-Garcia, and E. San Millán, "Fault Injection Methodologies," in *Radiation Effects on Integrated Circuits and Systems for Space Applications*, ed: Springer, 2019, pp. 127-144.
- [9] E. Cioroica et al., "Accelerated Simulated Fault Injection Testing," *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Toulouse, 2017, pp. 228-233.
- [10] Y. Nezzari and C. P. Bridges, "Modelling processor reliability using LLVM compiler fault injection," in *IEEE Aerospace Conference Proceedings*, 2018, pp. 1-10.
- [11] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "FAIL\*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *Dependable Computing Conference (EDCC), 2015 Eleventh European*, 2015, pp. 245-255.
- [12] H. Ziade, R. A. Ayoubi, and R. Velasco, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, pp. 171-186, 2004.
- [13] Y. Zhang, B. Liu, and Q. Zhou, "A dynamic software binary fault injection system for real-time embedded software," in *Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference on*, 2011, pp. 676-680.
- [14] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern

- computers," *IEEE Transactions on Software Engineering*, vol. 24, pp. 125-136, 1998.
- [15] D. Cotroneo and H. Madeira, "Introduction to Software Fault Injection," in *Innovative Technologies for Dependable OTS-Based Critical Systems*, ed: Springer, 2013, pp. 1-15.
- [16] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, 2003, pp. 137-143.
- [17] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "eDifferential fault injection on microarchitectural simulators," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, 2015, pp. 172-182.
- [18] E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, J. Tonfat, E. Macchione, et al., "Reliability on ARM processors against soft errors through SIHFT techniques," *IEEE Transactions on Nuclear Science*, vol. 63, pp. 2208-2216, 2016.
- [19] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1-6.
- [20] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, pp. 75-82, 1997.
- [21] J. Arlat, Y. Crouzet, and J.-C. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems," in *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, 1989, pp. 348-355.
- [22] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, 1989, pp. 340-347.
- [23] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger, "Application of three physical fault injection techniques to the experimental assessment of the MARS architecture," *Dependable Computing and Fault Tolerant Systems*, vol. 10, pp. 267-288, 1998.
- [24] A. Heinig, I. Korb, F. Schmoll, P. Marwedel, and M. Engel, "Fast and Low-Cost Instruction-Aware Fault Injection," in *Gl-Jahrestagung*, 2013, pp. 2548-2561.
- [25] E. van der Kouwe and A. S. Tanenbaum, "HSFI: accurate fault injection scalable to large code bases," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 144-155.
- [26] A. L. Sartor, P. H. Becker, and A. C. Beck, "A fast and accurate hybrid fault injection platform for transient and permanent faults," *Design Automation for Embedded Systems*, pp. 1-17, 2018.
- [27] J. Vankeirsbilck, T. Cauwelier, J. Van Waes, H. Hallez, and J. Boydens, "Software-Implemented Fault Injection for Physical and Simulated Embedded CPUs," in *2018 IEEE 27th International Scientific Conference Electronics, ET 2018 - Proceedings*, 2018.
- [28] L. Yin, Y. Ri-huang, B. Jian-wei, and Y. Chun-hui, "Research on a Software Fault Injection Model Based on Program Mutation," in *Information Science and Control Engineering (ICISCE), 2015 2nd International Conference on*, 2015, pp. 419-423.
- [29] M. M. Islam, N. M. Karunakaran, J. Haraldsson, F. Bernin, and J. Karlsson, "Binary-Level Fault Injection for AUTOSAR Systems (Short Paper)," in *2014 Tenth European Dependable Computing Conference*, 2014, pp. 138-141.
- [30] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys (CSUR)*, vol. 48, p. 44, 2016.
- [31] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A tool for the validation of system dependability properties," in *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, 1992, pp. 336-344.
- [32] T. K. Tsai, R. K. Iyer, and D. Jewitt, "An approach towards benchmarking of fault-tolerant commercial systems," in *Proceedings of Annual Symposium on Fault Tolerant Computing*, 1996, pp. 314-323.
- [33] P. Yuste, J. C. Ruiz, L. Lemus, and P. Gil, "Non-intrusive software-implemented fault injection in embedded systems," in *Latin-American Symposium on Dependable Computing*, 2003, pp. 23-38.
- [34] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: An intermediate code-level fault injection tool for hardware faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 11-16.
- [35] P. Bernardi, L. M. V. Bolzani, M. Rebaudengo, M. S. Reorda, F. L. Vargas, and M. Violante, "A new hybrid fault detection technique for systems-on-a-chip," *Ieee Transactions on Computers*, vol. 55, pp. 185-198, Feb 2006.
- [36] 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.
- [37] S. A. Asghari and H. Taheri, "An Effective Soft Error Detection Mechanism using Redundant Instructions," *International Arab Journal of Information Technology*, vol. 12, pp. 69-76, Jan 2015.
- [38] J. Vankeirsbilck, V. B. Thati, J. Van Waes, H. Hallez, and J. Boydens, "Control flow aware software-implemented fault injection for embedded CPUs," in *2017 XXVI International Scientific Conference Electronics (ET)*, 2017, pp. 1-4.
- [39] E. Chielle, F. L. Kastensmidt, and S. Cuena-Asensi, "Overhead Reduction in Data-Flow Software-Based Fault Tolerance Techniques," in *FPGAs and Parallel Architectures for Aerospace Applications*, ed: Springer, 2016, pp. 279-291.