IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 24 May 2023, accepted 15 June 2023, date of publication 19 June 2023, date of current version 26 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3287503

SURVEY

A Survey of QEMU-Based Fault Injection Tools & Techniques for Emulating Physical Faults

YOHANNES B. BEKELE, (Graduate Student Member, IEEE), DANIEL B. LIMBRICK, (Senior Member, IEEE), AND JOHN C. KELLY, (Member, IEEE)
Electrical and Computer Engineering Department, North Carolina Agricultural and Technical State University, Greensboro, NC 27411, USA
Corresponding author: Yohannes B. Bekele (ybbekele@aggies.ncat.edu)

ABSTRACT Fault Injection (FI) is a method used to quantify the reliability and resilience of a system by assessing the system's ability to detect, locate, and/or mitigate fault occurrences. At the architecture level, targeted bit flips at specific times and locations can help quantify the response of a running application to unwanted changes in state and memory values. FI campaigns of this type can be performed on the target hardware virtual implementations of the target device. In this paper, we present a survey of Quick EMUlator (QEMU) based FI techniques. After discussing the various techniques proposed by academia and industry, we classified them into categories and compare their attributes. This review will help researchers understand the capabilities and limitations of using the QEMU emulator for FI-based system reliability analysis. Additionally, we identify the gaps in existing techniques and propose opportunities for extensions.

INDEX TERMS Dependability, fault injection, QEMU, reliability, security, virtualization.

I. INTRODUCTION

Computing systems can experience faults from noise in the electrical system, electromagnetic interference (EMI), a strike to a chip surface from a particle, or radiations from the surrounding environment in the form of cosmic rays [1]. Faults that do not result in permanent functional damage to the system are called transient faults or soft errors. The occurrence of such faults is heightened by environmental conditions, such as the altitude and temperature of the system [2]. Owing to the increased operational frequency in recent technology, reduction in supply voltage, and decrease in technology node sizes, transistors' susceptibility to soft errors is increasing [3]. These errors can disrupt the functionality of computing systems, making dependability a concern.

Dependability is first introduced as a generalized concept that encompasses or consists of the attributes availability, integrity, reliability, maintainability, and safety [22]. In order to maximize the dependability of a system and to keep the correct operation of system components as per the required standard, fault tolerance techniques should be used. In order to put forward fault tolerance techniques a study of faults should be conducted at the hardware or software levels of a given system. The hardware level fault study, which is also the focus of this paper, has two main advantages as compared to its software counterpart. The first one is, output from the lower level of abstraction connects to the probability of a bit flip in software, leading to resilience approaches post-silicon and the second is that analysis of faults at application and architecture levels can give way to targeted hardening and selective node hardening approaches.

There are various methods that are utilized in fault tolerance studies. These include data collection and analysis of real-world fault occurrence scenarios of soft errors, laboratory fault injections to systems, and fault injection-based approaches. The first two approaches need specialized equipment which helps with fault injection and collection of the fault scenarios. In addition, laboratory fault injections result in damaging the system under question which makes them inappropriate choices for systems that are under operation. The occurrence of real faults, for instance from environmental sources, also takes a long time so collecting data that is needed for detailed analysis is a daunting task and time-consuming. These and other drawbacks in fault tolerance study techniques led academia and industry to rely on fault injection-based approaches. The FI-based tolerance study is

The associate editor coordinating the review of this manuscript and approving it for publication was Wenbing Zhao.

defined as ''a validation technique of the dependability of systems based on the realization of controlled experiments in order to evaluate the behavior of the systems in the presence of faults'' [28]. Although dependability can be evaluated through rigorous life-long testing, the required duration to carry out this testing and obtain a statistically sufficient number of failures makes it impractical for such implementations. This gap can be closed using FI campaigns that provide this information using as many as necessary FI campaigns that the implementer has full control on [43]. In this approach, a faulty situation is emulated on the system or component with each campaign round by injecting a fault into it and recording its fault-to-error conversion route and effects. This is then utilized to calculate the fault tolerance parameters.

Quick EMUlator (QEMU) is one of the platforms used for Simulation-based Fault Injection (SFI). QEMU is an open-source machine emulator which uses the Dynamic Binary Translation (DBT) of a given target CPU application code, utilizing various optimizations to keep execution speed as close as possible to native system execution. QEMU employs a two-step DBT approach, the first of which involves translating the target machine code into a machine-independent intermediate representation, which is then represented with the host machine code for execution. QEMU strives to be useful in a wide range of situations. The advantages that made QEMU the priority choice in developing tools for virtualization-based fault injection include:

- It allows unlimited modification of the codes used for the device for the inclusion of fault injection capability.
- It supports different processor architectures more than any virtualization platform currently.
- There is no need for any modification in the application under test or the need for the microprocessor design information. This characteristic assists in the emulation of complex systems.

In this paper, we complete a survey on QEMU-based fault injection techniques and approaches. We selected the papers based on the keywords *QEMU, virtualization, reliability, dependability, fault injection* and their combination. We categorize the techniques based on the components in a system that they are applied to, their fault modeling approaches, and their evaluation methodology. The paper is organized as follows. Sections II and III discuss fault occurrence in digital systems and provides an overview of fault injection approach. Section IV discusses QEMU-based fault injection approaches and points out the similarities and differences between the approaches. Section V categorizes the tools or frameworks discussed based on various metrics. Section VI indicates proposed future research directions that can further exploit QEMU features in FI works. Finally, Section VII Summarizes the survey and shows some areas that are open for research.

II. MODELING HARDWARE FAULTS

System reliability is greatly influenced or impacted by the fault-to-error conversion scenario and its consequence. A particle hit to the device or chip surface, often known as a transient fault, is one method of fault generation. A transient issue has a short-term impact on the system it affects and can either cause an error or be covered up through various masking strategies including electrical masking, logical masking, and latch window masking.

A. BASIC MECHANISMS OF PHYSICAL FAULTS

Single event transients (SETs) are transient voltage pulses generated at the output gates by a particle striking a chip surface [37]. The resulting error from a SET propagating to a memory element and stored in it is called a soft error. According to [3] soft errors are caused by two main factors: first, alpha particle emissions from radioactive contaminants on packing materials, and second, cosmic rays, which are high-energy rays that emanate from outer space and generate a complicated chain of secondary particles in the environment [36]. Bulk CMOS affecting radiations such as SETs and single event upsets (SEUs) are covered in [4], and the effect on recent FinFET technology nodes is shown in [5].

Other kinds of physical fault attacks include voltage fault injection in the form of voltage glitches, Laser Fault Injection (LFI), Electromagnetic fault injection (EMFI), and software-induced hardware faults. Voltage fault injection is an attack that focuses on creating disturbances in a stable power supply or distribution system causing misbehavior in the attacked system. ''This is the result of setup time violations that can cause incorrect data to be captured allowing an attacker to tamper with the regular control flow, e.g., by skipping instructions, influencing a branch decision, corrupting memory locations, or altering the result of an instruction or its side effects'' [7]. A type of this attack known as a voltage glitch is a disturbance that is induced in the power supply line which is a transient voltage drop within a very short duration. Glitch timing is mathematically computed as latency from a specific triggering event such as an I/O activity or power-up [8]. Voltage glitch-based fault injection techniques are inexpensive, simple, and typically don't call for expensive equipment. They can be accomplished remotely as well as with the target device in hand. The attacker must, however, have access to the device's power supply line in order to use this approach. Even the security enclaves of the Intel [9] and AMD [10] were broken via voltage glitching attacks utilizing the Teensy 4.0 board.

In LFI, the attacker injects a fault during a system operation by using a laser module which also gives the flexibility of precisely controlling the fault injection timing and position in the system. ''Compared to other injection approaches laser fault injection has the highest time and space resolution for the most efficient attack capability'' [11]. A typical LFI setup includes a laser source, an objective lens, a motorized positioning table, and a controlling device. A digital oscilloscope can be utilized to accurately coordinate laser activation with the device's execution of the target procedure [12].

LFI is a semi-invasive attack approach, which requires the chip packaging to be removed using mechanical or chemical de-capsulation techniques in order to expose the chip to the laser source. This is the biggest disadvantage since it is not always feasible to de-package the chip without destroying the circuitry or bonding wires. The injection is normally done on the backside of the chip, as the components are protected from the front side. This creates another challenge as the absorption depth of silicon varies for different wavelengths, and therefore, the silicon substrate might need to be thinned down to allow an attack [13].

EMFI is performed by generating a localized short-duration high-intensity electromagnetic pulse that induces currents within internal chip circuitry. The attacker generates a stable sinusoidal signal at a given frequency that injects a harmonic wave creating a parasitic signal [14]. Such a signal can bias the clock behavior or inject additional power directly and locally into the chip. Equipment for this type of EM injection usually consists of a motorized positioning table, a signal generation module, and an oscilloscope. Generally, the equipment consists of a high-voltage pulse generator and a coil with a ferrite core, serving as an injection probe. This method provides a good trade-off between cost and precision. Pulse injectors can be bought for a relatively inexpensive price [15] and there are also more powerful and precise equipment that are expensive to purchase. EMFI does not need a device decapsulation for the chips enclosed and as contrasted to voltage glitching, there is no need to attach any wires to the power supply.

Considering software-induced hardware faults, the most notable are rowhammer and CLKSCREW. A bit flip in the dynamic random-access memory (DRAM) is introduced by the system-level attack known as rowhammer, which may be used to escalate user privileges [16] or perform other attacks. Bit flips that are repeated and controlled pose a serious risk to system security. In a DRAM memory, hammering a row causes it to electrically interact with other rows, generating variations in voltage levels. This influences the next row and makes it discharge more quickly than normal; if the memory refresh interval is reached, a bit flip will happen [17]. A shrink in device technology used in DRAM construction has led to memory cells holding smaller charges, while also being closely placed. This proximity results in the cells being vulnerable to electromagnetic interactions among each other, leading to memory errors. About 85% of the DDR3 memory modules from different manufacturers tested in [17] were found to be vulnerable to the rowhammer attack. The attack is also effective in recent DDR4 modules as pointed out in [18]. CLKSCREW [19] is another software-induced hardware fault that exploits the bugs in a dynamic voltage and frequency scaling (DVFS) system. DVFS is a technique, in which the voltage and frequency of a system processor are scaled dynamically for power consumption minimization. CLKSCREW fault occurs if the system is overclocked by applying a higher frequency than that of the maximum designated system frequency or Undervoltage which is the result of applying a voltage value lower than the rated minimum voltage.

The occurrence of these faults is not deterministic in its very nature. In addition, their natural occurrence takes a long time to consider real-world scenarios for research purposes. Hence, FI techniques become the go-to approaches to emulate these faults.

B. FAULT INJECTION CHARACTERISTICS

- **Fault Models:** Fault models, models which are used to simulate real-world fault occurrence scenarios, are classified into intermittent, permanent, and transient fault models based on their persistence. In addition, a given fault model can map fault effects to temporal and spatial configurations depending on the abstraction level under consideration. Hence, the provision of adjustable temporal fault features, such as fault injection time, fault release time, and spatial fault properties, may be used to identify transient fault models. This allows the modeling of single and multiple transient faults in single and multiple bit-upsets in memory elements. This significantly helps in the modeling of faults impacting consecutive time intervals as well as the generalization of descriptions for faults affecting both consecutive and non-consecutive time intervals.
  Bit-flip fault models are utilized to simulate real-world physical fault attacks from a security perspective. Bit-flip is a method of mimicking faulty behavior by inverting the bit value of a component or system location such as a stored value in a memory element. In contrast, set and reset fault models simulate faulty scenarios by setting a bit to either a logical '0' or logical' 1' regardless of the initial fault-free value which is pessimistic because it results in the fault site changing its value during a fault injection. After the application of a bit-flip fault, the faulty value persists until it is overwritten. Other fault types include stuck-at and set/reset faults which emulate different scenarios in real-world fault attacks.
- **Fault Locations:** Faults occur in components that comprise a system, including datapath logic, registers, and memory. Temporary faults such as voltage changes, magnetic fields, and radiation cause CPU errors in various components of the CPU such as data registers, address registers, data-fetching units, control registers, and arithmetic logic units (ALUs). A fault occurring in the CPU causes a bit flip in data that is controlled and processed by the CPU. The effect of a bit-flip can range from having no effect on the execution of the program to a system downstate in extreme cases. Furthermore, it is challenging to pinpoint the root of the problem since the circumstances in which a bit flip takes place are the same as those in which a software fault modifies a value.
  Memory components are also system components that can get affected by faults. The effect can be on stored bits in memory or the memory controller which manages the

operation of the memory component. A particle strike on a memory element can result in both SEU and MEU by affecting single or multiple bits in the stored data in that specific location. Thus, FI frameworks or platforms consider both scenarios in simulating real-world fault occurrences. The MEU effect intensifies with the scaling down of process technologies as newer smaller technology cells have an increased impact due to proximity effects. Other system components that can be affected by faults or result in bit-flip include I/O components, interconnections between various system components, and data transfer mechanisms such as buses and controllers for these components.

C. FAULT OUTCOMES

Previous researchers have attempted to classify transient faults by their impact on the system [20], [21]. There are 4 major categories of outcomes that are the result of a fault in a system. These are:

- Silent data corruption (SDC): The execution result differed from the correct output without detection by the system and resulted in incorrect output.
- Detected an unrecoverable error (DUE): The fault effect cause the workload to have an error which triggers a fail-stop mechanism such as an exception that let the system detect the error.
- Hang: After fault injection, the execution did not finish within a given predefined time limit which is usually the time duration the execution takes in a non-faulty operation scenario, for example, because it faultily entered an infinite loop.
- Masked / no effect: The execution and output behaved the same way as in a run without fault injection.

III. FAULT INJECTION SIMULATION & EMULATION

FI techniques can be categorized into hardware fault injection (HFI), emulation-based fault injection (EFI), software fault injection (SWIFI), and simulation-based fault injection (SFI). In hardware-based fault injection techniques, an actual hardware or a hardware framework is needed to simulate the faulty condition. There are two main categories of hardware-based FI; fault injection with contact and fault injection without contact. FI with contact is based on the idea of perturbing the integrated circuits with faults introduced at the pins that emulate both external and internal faults [6]. Tools based on this approach include RIFLE [23], FOCUS [24] and MESSALINE [25]. Fault injection without contact is based on the idea that the injector has no direct physical contact with the design under test and an external source produces a physical phenomenon such as heavy ion radiation that interacts with the circuit and produces the faults. Tools based on this approach include FIST [26] and MARS [27]. These approaches are better in speed but are more difficult to implement and have higher costs with respect to both the equipment needed to induce a fault and the resulting damage to the hardware being tested [6].

Emulation-based FI techniques combine hardware and software-based approaches to gain better speed and accuracy. They do not need any special facility making them more cost-effective and there is no limitation in choosing the fault locations. It is possible to validate the circuit in the initial steps of the design. One of the most popular techniques in the emulation-based fault injection approach is emulating the behavior of the circuit using FPGAs in the presence of faults [28]. They can be implemented using hardware reconfiguration-based approaches or instrumentation-based approaches. In the first scenario, faults are introduced into the process by partially reconfiguring the prototype as it is being executed. In the second scenario, the circuit is altered before it is implemented on the re-configurable hardware so that the errors in the chosen model can be introduced into the program while it is being executed. Some example tools from this category of fault injection include NETFI [29], a tool that injects faults at the register-transfer level of a processor by adding extra hardware to the sensitive registers of a target processor, and [30], a tool having three different techniques called time-multiplexed, state-scan and mask-scan, which offer different trade-offs between area overhead and performance. These two tools use the instrumentation-based approach of EFI. Whereas, Fault Tolerant-University of Seville Hardware DEbugging System (FT-UNSHADES) [31], which implements a read-modify-write approach, and its modified version FT-UNSHADES2 [32] which speeds up the communications by using PCIExpress, rather than USB transactions are examples of tools that implement hardware reconfiguration-based approaches.

If an analysis is proposed to be done without the need for special-purpose HW, SWIFI, and SFI tools can be used. SWIFI is accomplished by modifying the software executing on the system that the analysis is carried on. These modifications are errors inserted during compile time or run time. The ones based on errors during compile time introduce the faults into the source code or the assembly code of the program under test. In the case of faults inserted during run time, a trigger mechanism is necessary to insert the faults. The usage of SWIFI tools overcomes the cost, controllability, and repeatability issues of HFI. SWIFI modifies the contents of registers and memory elements to emulate the effect of real-world hardware faults. Notable SWIFI tools include FERRARI [33], FTAPE [34] and XCEPTION [35]. The drawbacks of SWIFI techniques include, accurately reproducing the faulty behaviors of the actual hardware and their definition for either specific operating systems or application programming interfaces (API).

These drawbacks of SWIFIs are addressed by SFI tools. SFI involves simulating the Design Under Test(DUT) using a Hardware Description Language (HDL) and then injecting faults into the simulated version of the design using software. This can be accomplished by modifying the high-level description of the target design with a saboteur module, which is in charge of the fault injection process, and by using the built-in commands of a simulator to inject errors into the

TABLE 1. Categories of FI techniques.

Category	Execution Domain
Hardware	Actual system component
Emulation	Software & Hardware approaches for speed-up & accuracy
Simulation	Using HDLs for DUT simulation
Software	Application code alteration

simulation of the design and not in the hardware description of the design itself [40]. Notable tools in SFI include MEFISTO [41] and VERIFY [42]. These SFI approaches should do a cost-benefit analysis of the trade-off between analysis accuracy and simulation performance. Table 1 summarizes FI technique categories.

IV. QEMU-BASED FAULT INJECTION

The development of fault injection simulators based on the QEMU platform allows designers to emulate soft errors and verify the efficiency of fault tolerance solutions with low overhead and higher repeatability. Due to the advantages mentioned in the previous sections, the use of QEMU for studying the effect of soft errors is increasing among the research community. In this section, the works that leverage QEMU features to inject faults in system components and emulate soft errors are discussed.

FAIL* [39] is an architecture-level FI framework for continuously assessing and quantifying fault tolerance in iterative software development processes. It is a FI tool for testing and quantifying software-based fault tolerance mechanisms deployed in software systems. It supports three simulators; Bochs, QEMU, and Gem5 emulating x86 and ARM architectures. Looking at its architecture, FAIL* is organized in a client/server architecture. It extends the back-end code of instances. This gives the tool interception and control power on the back-end execution in addition to its access to the simulated system state. The Campaign Controller component distributes parameter sets to the available FAIL* instances that it has received from a user-defined campaign. Each FI experiment uses a parameter set and uses the execution-environment abstraction (EEA) layer to control the target back end. By adding an interface module to this abstraction, actual target backends may be switched. The author's tested the tool on dOSEK OS [79] to show how it helps in helping the developer to converge to an optimally protected software stack.

Another QEMU-based FI platform that permits soft error introduction in emulated machine instructions is called F-SEFI [44]. F-SEFI acts during the tiny code generation (TCG) step and provides the structure to emulate soft errors by corrupting data at run-time intercepting instructions and replacing them with fault-injected versions. The F-SEFI broker is loaded dynamically after the QEMU hypervisor starts a virtual machine image. Without altering the source code of the application or the OS running on the kernel, F-SEFI intercepts instructions sent by applications operating within a guest OS, possibly corrupt them and then sends them to the host kernel. The F-SEFI tool is comprised of five major components: profiler, configurator, probe, injector, and tracker. Faults can be injected into the combinational logic of the CPU, register, and memory modules. The tool is demonstrated for successful fault injection into three benchmark applications: fast Fourier transform (FFT), Bit Matrix Multiplication, and K-Means Clustering. This tool is extended in [45], dubbed parallel-FSEFI (P-FSEFI), to handle parallel application execution.

In addition to the features of F-SEFI, P-FSEFI has the following features:

- Tools to configure, launch, and manage multiple sequential instances of F-SEFI.
- An extended, pluggable, and flexible interface for inserting fault models so that a user can develop their own fault injection techniques.
- The addition of permanent, high locality persistence, and low locality persistence faults.
- The ability to inject faults into parallel programs running on multiple virtual machines on the same, or different physical hosts [45].

Considering performance, the QEMU emulation slows down the application by about 10x, P-SEFI instrumentation slows an additional 30x, and the virtual machine (VM) distributing across multiple nodes adds some 1.06x slowdown on top. Overall, P-SEFI runs about 300x slower than under no emulation on the host hardware.

In [61] the authors leveraged dynamic binary instrumentation in a virtual machine-based fault injection environment to emulate soft errors and study the impact on the behavior of applications by proposing Chaser, a framework that supports two well-known virtual machines; QEMU and DECAF [62]. Chaser offers just-in-time (JIT) fault injection, the ability to trace the fault propagation path and programmable interfaces that can be modified easily. The design of Chaser is based on the P-FSEFI tool, [45] as an add-on function. Chaser has two main components; a JIT fault injector and a fault propagation tracer. The JIT fault injector component injects faults into the target process using instructions that are marked as target instructions by the tool user. The tool then exports its fault injection capabilities as interfaces that can be utilized by the user to define customized fault models. The fault propagation tracer traces the propagation of faults through a dynamic tainting technique. It leverages DECAF's bit-wise tainting [62] which is extended to support floating point instruction tainting. Tainted memory access activity is recorded by Chaser. Because instruction level tainting is not taken into account in this approach, an appropriate amount of fault propagation tracing completeness is sacrificed for the sake of a little performance penalty. The authors implemented three fault injectors described in F-SEFI [44] - a probabilistic injector, a deterministic injector, and a group injector. The tool is tested on various benchmarks from the Rodinia benchmark suite including bfs, kmeans, and lud, Matvec, and CLAMR.

MH-QEMU, a memory fault injector that is memory state-aware and implemented by extending a VM to intercept memory accesses, is introduced in [47]. MH-QEMU follows the same method as F-SEFI, [44], the only difference between MH-QEMU and F-SEFI is that MH-QEMU focuses on memory module faults that define memory state and memory access patterns. It has features that can be leveraged to analyze memory access behaviors or patterns such as a physical-to-virtual memory address mapping functionality in memory controllers in real-time scenarios. MH-QEMU consists of the following three modules, which are illustrated in Fig. 1: (1) a memory-access handler (MH), a user-defined handler function that can be registered and used as the hook to load and store accesses to the target VM's memory space, (2) fault injection scheduler (FS), which manages the MM and the MH by adhering to a scenario file that describes the time of fault injection and configurations of MH, and (3) memory mapper (MM), which is used to access the VM's memory from the host environment. The address-data mapper (ADM) obtains data about the guest OS, including information about processes and the memory page table. Via an API, a user can utilize the ADM from the MH. The target VM's configuration script may additionally contact the ADM to initialize additional MH-QEMU modules. With all the functionalities of FI in place, MH-QEMU performance evaluation shows that it is up to 3.4 times slower than a typical native QEMU implementation. Narrowing down the monitored memory region can reduce the overhead obtained [47].

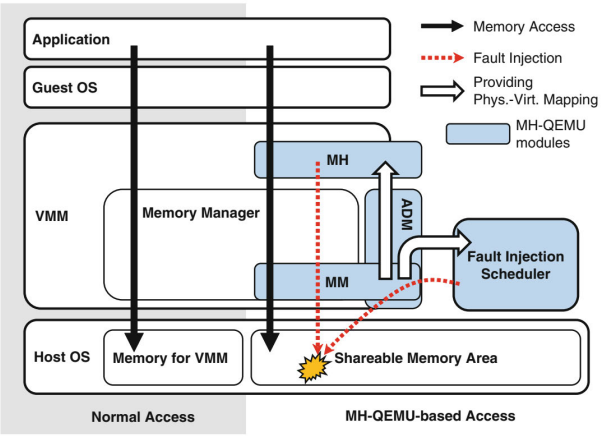

FIGURE 1. MH-QEMU Architecture. "It gathers memory access patterns, analyses them to create an appropriate fault injection plan, and applies it to target VM memory which is done from the host to avoid side effects to the target system" [47].

D-Cloud/FaultVM [46] is another fault injector that uses QEMU for checking the reliability of the memory, hard disk, and network controller system components. It is a software testing environment for parallel and distributed systems leveraging cloud computing technology. By interpreting system setup and test scenarios defined in XML in the D-Cloud front-end, it enables test procedure automation employing several cloud computing resources in a cloud computing environment. Moreover, D-Cloud makes it possible to test the hardware flaws by flexibly simulating hardware flaws with FaultVM. By modifying QEMU, the fault injector of FaultVM and the hardware are implemented at the same layer. The fault is injected in accordance with the injection command when it arrives at FaultVM through the network from the D-Cloud front end.

In [48] another QEMU-based FI methodology that emulates soft errors is proposed which accelerates the analysis and debugging of complex systems and facilitates validation of fault tolerance techniques. There are two levels of simulation, the bootloader level, which damages the initialization sequence and causes failures in the basic elements of the system, and a system level fault injection which defines faults injected in the operating system and is used to observe problems in the execution of applications and hardware management. A case study on an x-86 processor real-time operating system (RTEMS) and benchmarks such as matrix multiplication, sha1, and quicksort showed the vulnerability level of general purpose registers (GPR) to faults and which in turn can be leveraged by designers in evaluating fault tolerance technique implementations at the software level.

Another method for abstracting various types of hardware failure models within QEMU is proposed in [49]. It defines a simulation environment that simulates hardware faults in early dependability analysis of embedded software (ESW) applications. The approach adopts a single fault analysis using stuck-at, transient, and delay scenarios with multi-bit faults being considered by a different fault in the system. The approach disables the caching mechanism in QEMU and this degrades the performance of the underlying system. Nevertheless, the tool's accuracy in injecting faults in registers is shown to be good. This technique exhibits comparable characteristics in simulation runs from the perspective of dependability analysis accuracy compared to register-transfer level (RTL) failure simulation, but it is quicker, as demonstrated by experimental findings on bsearch, tcas, mandelbot, and dhrystone benchmark applications [49].

A fault injection framework, called BitVaSim, which requires a low overhead in comparison to a fault-free QEMU execution, is described in [50]. The framework is designed for embedded development boards powered by PowerPC or ARM processors and has a built-in test (BIT) software operating environment. BitVaSim operates as a simulator which results in a smooth operation that does not affect the underlying hardware or software systems. The reachability of its simulated parts makes the integration of additional fault models an easily reachable task. The framework uses key-value pair abstractions to describe fault modes that the simulator is utilizing to carry out its functions. Utilizing these fault modes, it simulates the given hardware board in order to monitor the activation and impact of these faults on the hardware system in question focusing on the BIT system behavior in detail. A modified version of QEMU that has five additional modules built into it is utilized in BitVaSim. The modeling and configuration module, the fault injector module, the monitor module, and the feedback module are the additional modules. Due to the fault injection functionality,

BitVaSim with the DBT exhibits a performance drop of 5% to 7% when compared to a native QEMU environment execution performance. A similar work is also presented in [65].

The work, [51], presented FIG-QEMU (fault injection by GDB for QEMU), an automated fault injection system that simulates a variety of single event effects. The paper proposes a hardware-implemented fault injection system based on QEMU and evaluates the robustness of the target OS and application software against soft errors from the CPU. When we look at its implementation approach, this work uses the approach of [52] to inject faults into the user-visible registers and memory units of the PowerPC750 virtual machine, and the side effects of the faults at the application level are evaluated. The difference between [52] and FIG-QEMU is that FIG-QEMU can accept the source-free executable as the input of the system, so as to inject faults without the source code. FIG-QEMU has six main components; main controller, fault-lib, trigger, monitor, injector, and collector. It is based on Python extended GDB interface, so it has good performance and portability and can run on the target platform only with a small modification. The authors conducted a total of 6.3 million single-bit flip fault simulation experiments, among which 6.21% of the register faults and 3.53% of the memory faults caused system crashes respectively. The main drawback is that FIG-QEMU cannot be used on platforms that do not support GDB debugging tools because it relies on GDB.

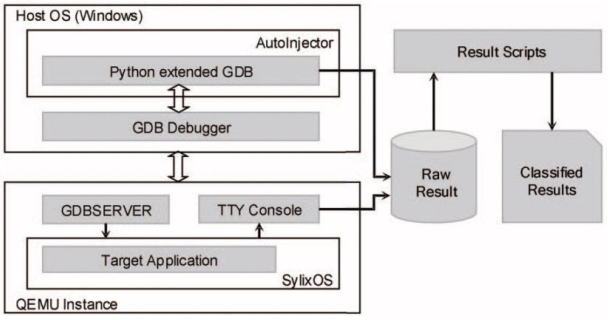

FIGURE 2. FIG-QEMU Architecture. FIG-QEMU does FI via GDB debugger without secondary exploration of QEMU [51].

Höller et. al presented a Fault Injection framework, FIES, for the evaluation of software-based self-tests (SBSTs) [54] according to the safety standard IEC 61508 [67]. This framework is compatible with commonly used embedded commercial off-the-shelf (COTS) processors and offers analytical feedback about the diagnostic coverage of self-tests at the early design phases. It simulates faults in an ARM processor's control and execution paths and includes an expanded fault model to mimic memory coupling problems. The suggested approach evaluates SBSTs by hardly altering them in order to raise the number of discovered defects. Findings are presented as a concise list of the found flaws and the associated diagnostic coverage, which demonstrates the caliber of the SBST under examination. The framework may also be utilized in the early phases of design since a hardware prototype is not necessary for its utilization. The framework allows for the interactive specification of automatically inserted defects in XML. The authors used the evaluation of a memory test to show the framework's performance and applicability.

In [63] a virtual FI framework that simulates safety-standard aligned fault models and supports COTS software implementations, as well as popular edge device embedded processors is presented. The work also shows the procedure for integrating the framework into various software development stages. The framework extends the FI tool presented in [54] which is also based on QEMU. It supports advanced memory and processor fault models that emulate real-world fault occurrence scenarios. The tool proposes a reliability assessment based on four steps: hardware usage characteristics profiling, fault library creation, fault injection and saving resulting application outputs, and interpretation and reporting of outputs. A similar work is also presented by Höller et.al. [64].

In [55] and [56] the authors presented a QEMU fault injector (QEFI), which enhances service availability by assessment of a computer system behavior in the presence of memory faults and a method of handling exceptions caused by disturbances in executable code. QEFI is designed as a tool for system-wide and kernel-based FI. The chosen approach allows the creation of a fault injection framework focused on simulating hardware faults and testing software reactions to them. Faults can be triggered and injected in CPU, RAM, and peripherals system components in a user-defined probability. Its three major parts are the fault injection control framework, the QEMU emulator, and a fault injection control library used as a proxy between the two other layers, responsible mainly for communication and QEMU execution management. The authors made modifications to TCG in QEMU which may affect the speed of QEMU. In addition, QEFI supports FI through GDB, and to integrate QEMU FI execution framework and GDB, the support for GDB server protocol that is already present in QEMU has been used. The protocol has been enhanced to support not only breakpoints and watch points but also injection points. The quality of this approach is legitimized by the works presented in [57] where the same real device fault injection framework was run on a real device and an emulated one with similar results.

The work by Ferraretto and Pravadell presented an approach for simulating hardware faults on CPU system components [58]. Fault abstractions are used for permanent and transient fault models that preserve the quality of software dependability analysis. The DBT feature of QEMU is leveraged by this approach in order to minimize the impact on the performance of the fault injection procedure on the emulator. Faults can be injected into GPRs, instruction registers (IR), and program status registers (PSR). The experiment has been conducted on three benchmarks; btrees, mandelbrot, and dhrystone. The fault injection mechanism that is developed in this work is shown in Fig. 3.

Amarnath et al. proposed a FI framework that can be used to emulate OS fault propagation that injects random hardware

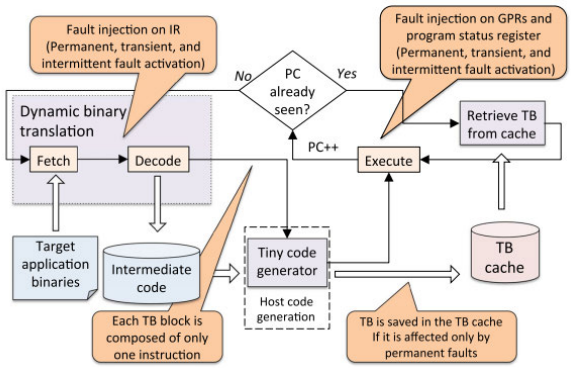

FIGURE 3. Simulation-based Fault Injection with QEMU Architecture. "The approach minimizes the impact of the fault injection procedure on the emulator performance by preserving the original DBT mechanism in QEMU" [58].

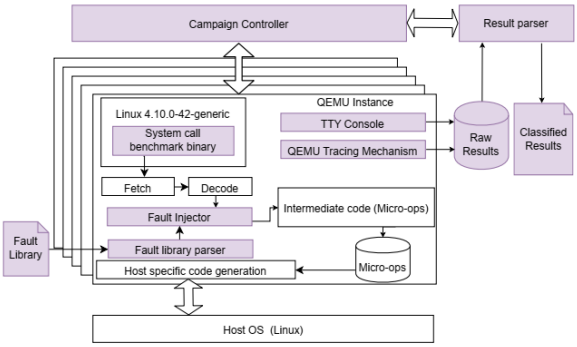

FIGURE 4. Schematic overview of the FI framework proposed in [53]. "The implementation dynamically reduces the translation block size to one instruction only when injecting the fault and otherwise keeps this speedup feature of QEMU enabled" [53].

faults into the CPU and measures the span these faults propagate and surface as application-level side effects [53]. This QEMU-based framework simulates bit flips in x86 general and special purpose registers during the execution of system calls of Linux 4.10 and classifies the injection effects at the application level. The approach extends QEMU version 2.9.1 [59]. It modifies a given register's value with FI before the intermediate DBT stage to inject a specific user or framework-selected fault from the fault library. Using Linux kernel test programs, the authors assessed and categorized the consequences of soft errors inserted during the execution of the clone, futex, mmap, mprotect, and pipe syscalls. Results reveal that, on average, 76.3% of the injected faults are benign and do not appear at the application level as shown by carrying out 4.48 million FI simulations for various syscalls on the x86 architecture.

In [60], the authors presented the soft error fault injection (SEFI) framework, a software system profiling framework for soft error vulnerability assessment. In particular, the paper focuses on logic soft error injection. The authors used QEMU to demonstrate the modification of emulated machine instructions in order to introduce hardware faults as soft errors. With this technique, the paper shows the possibility and feasibility of injecting soft error simulations in the logic operations of a

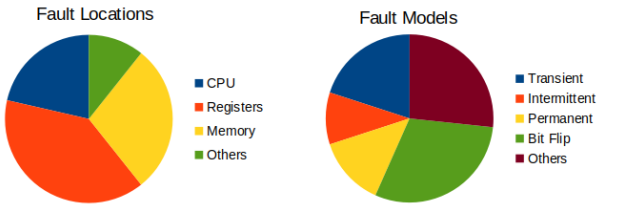

FIGURE 5. Distribution of surveyed works' focus areas.

target application without impact on other applications or the underlying operating system. SEFI operates in three steps.

1) Booting guest environment and starting FI application.
2) The intended target application's code region is probed by the guest OS, which then alerts the virtual machine which code regions to monitor.
3) Application is released, allowing it to run.

The VM then monitors the system instructions running on the given machine and supplements the ones it identifies as crucial.

In [66], Wanner et al. designed VarEMU, a framework for QEMU extension for assessing variability-aware software methods. VarEMU gives system users a platform to simulate variations in system power consumption and fault characteristics so they can detect and adjust to these changes in the software. Fault injections have the possibility to be carried out before or after, or completely replace the execution of any instruction with precise control over faults. Fault injection is done through a guest OS system call. Faults are injected using an imported library that can interface with system calls. The framework is used to inject faults in register components of a system CPU.

V. CATEGORIZATION OF QEMU-BASED FAULT INJECTION APPROACHES

The previous section provided a general summary of the works that leverage various features of QEMU for fault injection emulation experiments. In this section, the works are categorized based on:

- The components that are affected by FI or tested for reliability
- The fault modeling approach taken
- The evaluation methodology used to evaluate the performance or functionality of the tool

A. COMPONENTS TESTED FOR RELIABILITY

When it comes to the types of components tested for reliability or fault being injected, generally, the components can fall into the following categories:

- CPU - Here the CPU components which include functional units [44], logic units [45], instruction decoders [54], and related components are tested [45], [55].
- Memory - The memory cells such as RAM and related parts are tested [44], [45], [46], [47], [50], [51], [54], [55], [63].

TABLE 2. Summary of QEMU-based FI tools and techniques.

	Fault Location				Fault Model				
Methods	CPU	Register	Memory	Others	Transient	Intermittent	Permanent	Bit-Flip	Others
FAIL* [39]				✓	✓			✓	
FSEFI [44]	✓	✓	✓					✓	
P-FSEFI [45]	✓	✓	✓				✓	✓	
MH-QEMU [47]			✓						✓
Geissler et. al., [48]		✓			✓	✓	✓		
Guglielmo et.al., [49]		✓			✓				✓
BitVaSim [50]		✓	✓	✓				✓	✓
FIG-QEMU [51]		✓	✓					✓	
FIES [54]	✓	✓	✓					✓	✓
QEFI [55]		✓	✓	✓					✓
Ferraretto et.al., [58]		✓			✓	✓	✓		✓
Amarnath et.al., [53]		✓			✓			✓	
SEFI [60]	✓							✓	
Chaser [61]	✓							✓	
Höller et.al., [63]	✓		✓		✓	✓	✓		✓
VarEmu [66]		✓							✓

- Registers - General purpose and special purpose registers are tested in [44], [45], [48], [49], [50], [51], [53], [54], [58], and [66].

In addition to these components; instructions, network interfaces, controllers for the hard disk, USB, and network and block devices are tested.

B. FAULT MODELING APPROACH

Although the main aim of all the tools discussed here is to induce faults via FI, they differ in the faults or fault models they have taken. These can be broadly classified as

- Transient [39], [48], [49], [53], [58], [63]
- Intermittent [48], [58], [63]
- Permanent [45], [48], [49], [50], [54], [58], [63], [66] faults.

There are faults such as packet drop and packet modification [55], memory-state-aware faults [47], and faulty instructions [44] used by different tools.

C. EVALUATION METHODOLOGY

The evaluation methodologies used in the FI tools or frameworks basically serve two purposes; evaluate the functionality of the tool and evaluate the performance characteristics of the same. These methodologies generally fall into two main categories. These are:

- Standard benchmark applications used by academia and industry and available free of cost or commercially such as NAS Parallel benchmark suite [68], MiBench benchmark suite [69].
- In-house or industry-specific benchmarks developed by the tool authors themselves or acquired from a third party.

Table 2 shows the type of components tested and the fault modeling approaches taken by the tools discussed in Section IV. The tools or approaches surveyed span a range in the type of components that they evaluate and the methodology they employ. Hence, a user can select from those tools based on the specific application requirements. In using the tools, if there is no need for a specific need for functionality, using the original tools is recommended as there are additional overheads incurred by the modified tools. For instance, [63] is an extension of [54] with additional overhead and support for advanced memory and processor fault models. Similarly, [61] is an add-on functionality on [44]

with an additional overhead of about 10X of normal QEMU environment and JIT fault injection added functionality. If an application is a parallel or multi-thread application, [45] is a choice for physical fault emulation.

Fig. 5 shows the distributions of fault locations and fault models in the surveyed works. As can be seen, the recent trend in QEMU-based FI tools and techniques is on register system component fault injections and using bit flips as prominent fault models.

VI. FUTURE RESEARCH DIRECTIONS

Although the tools or frameworks developed so far went a long way in attaining this goal, there are still potential areas to enhance these tools or develop new tools with the same purpose but using more robust approaches. These areas include:

- **Parallel applications' reliability analysis:** There are a few tools that are used to assess the reliability of parallel applications and high-performance computing (HPC). Of the tools developed so far, only P-FSEFI [45] uses QEMU-based FI on parallel applications. As parallel execution of applications is becoming dominant, devising ways to assess their reliability leveraging QEMU features can be one huge research direction.
- **Reliability analysis of hardware accelerators:** Currently, hardware accelerators such as FPGAs are in widespread use for performance enhancement and are part of major projects and data warehouses. Hence, reliability analysis of these accelerators and their components using QEMU-based FI is also one area to consider for further research.
- **Performance overhead improvement:** Although the discussed tools meet the main criteria of FI for reliability analysis, most of them cause severe performance degradation on the systems they are deployed on. For instance, P-FSEFI runs at about 30X slower than QEMU run [45] and SASSIFI [80] showed a 1.02× to 166× slowdown at the application level and a 5.2× to 488× at the kernel level. Thus one major area to focus on is developing tools with the same purpose and with less performance overhead or modifying existing tools by enhancing their operational performance and reducing overheads.
- **Comparison with approaches at other layers:** There are FI techniques at other layers of a system and a comparison study to performance metrics for comparing fault handling at the software or hardware level can be one area of focus.

VII. SUMMARY

This work is a survey of virtualization-based fault injection tools and techniques focusing on those based on the QEMU platform. As can be seen from the discussion on the tools or frameworks, there are various approaches to attain the goal of reliability analysis using QEMU-based FI. The works also show the progress in performance and other metrics in using QEMU for FI-based reliability analysis. We also tried to point out future potential areas of research in expanding the use and improving the performance of these techniques.

REFERENCES

[1] N. Seifert, X. Zhu, and L. W. Massengill, "Impact of scaling on soft-error rates in commercial microprocessors," *IEEE Trans. Nucl. Sci.*, vol. 49, no. 6, pp. 3100–3106, Dec. 2002.

[2] N. Seifert, B. Gill, K. Foley, and P. Relangi, "Multi-cell upset probabilities of 45 nm high-k + metal gate SRAM devices in terrestrial and space environments," in *Proc. IEEE Int. Rel. Phys. Symp.*, Apr. 2008, pp. 181–186.

[3] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 305–316, Sep. 2005, doi: 10.1109/TDMR.2005.853449.

[4] G. Hubert and L. Artola, "Single-event transient modeling in a 65-nm bulk CMOS technology based on multi-physical approach and electrical simulations," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 6, pp. 4421–4429, Dec. 2013, doi: 10.1109/TNS.2013.2287299.

[5] G. Hubert, L. Artola, and D. Regis, "Impact of scaling on the soft error sensitivity of bulk, FDSOI and FinFET technologies due to atmospheric radiation," *Integration*, vol. 50, pp. 39–47, Jun. 2015, doi: 10.1016/j.vlsi.2015.01.003.

[6] H. M. Quinn, D. A. Black, W. H. Robinson, and S. P. Buchner, "Fault simulation and emulation tools to augment radiation-hardness assurance testing," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 3, pp. 2119–2142, Jun. 2013.

[7] L. Zussa, J. Dutertre, J. Clediere, and B. Robisson, "Analysis of the fault injection mechanism related to negative and positive power supply glitches using an on-chip voltmeter," in *Proc. IEEE Int. Symp. Hardware-Oriented Secur. Trust (HOST)*, Arlington, VA, USA, May 2014, pp. 130–135.

[8] C. Bozzato, R. Focardi, and F. Palmarini, "Shaping the glitch: Optimizing voltage fault injection attacks," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, no. 2, pp. 199–224, Feb. 2019.

[9] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia, "VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface," in *Proc. 30th USENIX Secur. Symp. (USENIX Security)*, 2021, pp. 699–716.

[10] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert, "One glitch to rule them all: Fault injection attacks against AMD's secure encrypted virtualization," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 2875–2889.

[11] K. Matsuda, S. Tada, M. Nagata, Y. Komano, Y. Li, T. Sugawara, and M. Iwamoto, "An IC-level countermeasure against laser fault injection attack by information leakage sensing based on laser-induced optoelectric bulk current density," *Jpn. J. Appl. Phys.*, vol. 59, Feb. 2019, Art. no. SGGL02, doi: 10.7567/1347-4065/ab65d3.

[12] J. Breier and X. Hou, "How practical are fault injection attacks, really?" *IEEE Access*, vol. 10, pp. 113122–113130, 2022, doi: 10.1109/ACCESS.2022.3217212.

[13] J. Breier and C. Chen, "On determining optimal parameters for testing devices against laser fault attacks," in *Proc. Int. Symp. Integr. Circuits (ISIC)*, Dec. 2016, pp. 1–4.

[14] Y. Hayashi, N. Homma, T. Sugawara, T. Mizuki, T. Aoki, and H. Sone, "Non-invasive EMI-based fault injection attack against cryptographic modules," in *Proc. IEEE Int. Symp. Electromagn. Compat.*, Aug. 2011, pp. 763–767.

[15] C. O'Flynn, "MINimum failure: EMFI attacks against USB stacks," in *Proc. 13th USENIX Workshop Offensive Technol. (WOOT)*, 2019, pp. 1–10.

[16] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges: How to cause and exploit single bit errors," in *Proc. BlackHat*, 2015, pp. 1–71.

[17] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 361–372.

[18] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Reverse engineering Intel DRAM addressing and exploitation," 2015, *arXiv:1511.08756*.

[19] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *Proc. 26th USENIX Secur. Symp. (USENIX Security)*. Vancouver, BC, Canada: USENIX Association, 2017, pp. 1057–1074.

[20] H. T. Nguyen, Y. Yagil, N. Seifert, and M. Reitsma, "Chip-level soft error estimation method," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 365–381, Sep. 2005, doi: 10.1109/TDMR.2005.858334.
[21] S. Mukherjee, *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann, 2008.
[22] K. Rozier, *Dependability Management—Part 1: Dependability Management Systems*, International Electrotechnical Commission, IEC Standard EN 60300-1:2003, 2003.
[23] H. Madeira, M. Rela, F. Moreira, and J. G. Silva, "RIFLE: A general purpose pin-level fault injector," in *Proc. 1st Eur. Dependable Comput. Conf. (EDCC-1)*. Berlin, Germany: Springer-Verlag, 1994, pp. 199–216.
[24] G. S. Choi and R. K. Iyer, "FOCUS: An experimental environment for fault sensitivity analysis," *IEEE Trans. Comput.*, vol. 41, no. 12, pp. 1515–1526, 1992, doi: 10.1109/12.214660.
[25] J. Arlat, "Validation de la Sûreté de fonctionnement par injection de fautes. Méthode mise en OEuvre et application," Ph.D. thesis, LAAS-CNRS, Toulouse, France, Dec. 1990.
[26] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *Proc. 19th Int. Symp. Fault-Tolerant Computing. Dig. Papers*, Los Alamitos, CA, USA, 1989, pp. 340–347.
[27] J. Karlsson, J. Arlat, and G. Leber, "Application of three physical fault injection techniques to the experimental assessment of the MARS architecture," in *Proc. 5th Annu. IEEE Int. Working Conf. Dependable Comput. Crit. Appl.* Los Alamitos, CA, USA: IEEE CS Press, 1995, pp. 150–161.
[28] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems," in *Proc. 9th IEEE Int. Conf. Design Technol. Integr. Syst. Nanosc. Era (DTIS)*, May 2014, pp. 1–6, doi: 10.1109/DTIS.2014.6850649.
[29] W. Mansour and R. Velazco, "An automated SEU fault-injection method and tool for HDL-based designs," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 4, pp. 2728–2733, Aug. 2013.
[30] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia, and L. Entrena, "Autonomous fault emulation: A new FPGA-based acceleration system for hardness evaluation," *IEEE Trans. Nucl. Sci.*, vol. 54, no. 1, pp. 252–261, Feb. 2007.
[31] M. A. Aguirre, D. Merodio, J. N. Tombs, F. Munoz, V. Baena, H. Guzman, J. Napoles, A. Torralba, A. Fernandez-Leon, and F. Tortosa-Lopez, "Selective protection analysis using a SEU emulator: Testing protocol and case study over the Leon2 processor," *IEEE Trans. Nucl. Sci.*, vol. 54, no. 4, pp. 951–956, Aug. 2007.
[32] J. M. Mogollon, H. Guzmán-Miranda, J. Nápoles, J. Barrientos, and M. A. Aguirre, "FTUNSHADES2: A novel platform for early evaluation of robustness against SEE," in *Proc. 12th Eur. Conf. Radiat. Effects Compon. Syst.*, Sep. 2011, pp. 169–174.
[33] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A tool for the validation of system dependability properties," in *Proc. 22nd Annu. Int. Symp. Fault-Tolerant Comput.* Los Alamitos, CA, USA: IEEE CS Press, 1992, pp. 336–344.
[34] D. T. Stott, Z. Kalbarczyk, and R. K. Iyer "Using NFTAPE for rapid development of automated fault injection experiments," Center Reliable High-Perform. Comput., Univ. Illinois Urbana Champaign, Champaign, IL, USA, Res. Rep., 1999.
[35] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 125–136, Feb. 1998, doi: 10.1109/32.666826.
[36] M. Eslami, B. Ghavami, M. Raji, and A. Mahani, "A survey on fault injection methods of digital integrated circuits," *Integration*, vol. 71, pp. 154–163, Mar. 2020, doi: 10.1016/j.vlsi.2019.11.006.
[37] U. Kretzschmar, A. Astarloa, J. Jiménez, M. Garay, and J. D. Ser, "Fast and accurate single bit error injection into SRAM based FPGAs," in *Proc. 22nd Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2012, pp. 675–678.
[38] *QEMU*. Accessed: Nov. 5, 2022. [Online]. Available: http://wiki.QEMU.org
[39] 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 *Proc. 11th Eur. Dependable Comput. Conf. (EDCC)*, Sep. 2015, pp. 245–255.
[40] Ó. Ruano, F. García-Herrero, L. A. Aranda, A. Sánchez-Macián, L. Rodriguez, and J. A. Maestro, "Fault injection emulation for systems in FPGAs: Tools, techniques and methodology, a tutorial," *Sensors*, vol. 21, no. 4, p. 1392, Feb. 2021, doi: 10.3390/s21041392.
[41] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: The MEFISTO tool," in *Predictably Dependable Computing Systems* (ESPRIT Basic Research Series), B. Randell, J. C. Laprie, H. Kopetz, and B. Littlewood, Eds. Berlin, Germany: Springer, 1995.
[42] V. Sieh, O. Tschache, and F. Balbach, "VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions," in *Proc. IEEE 27th Int. Symp. Fault Tolerant Comput.*, Seattle, WA, USA, Jun. 1997, pp. 32–36.
[43] L. Berrojo, I. Gonzalez, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. Lopez, "New techniques for speeding-up fault-injection campaigns," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, 2002, pp. 847–852.
[44] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-SEFI: A fine-grained soft error fault injection tool for profiling application vulnerability," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 1245–1254.
[45] Q. Guan, N. Bebardeleben, P. Wu, S. Eidenbenz, S. Blanchard, L. Monroe, E. Baseman, and L. Tan, "Design, use, and evaluation of P-FSEFI: A parallel soft error fault injection framework for emulating soft errors in parallel applications," in *Proc. SPIE9th EAI Int. Conf. Simul. Tools Techn. (SIMUTOOLS)*. Brussels, Belgium: Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, 2016, pp. 9–17.
[46] T. Hanawa, H. Koizumi, T. Banzai, M. Sato, S. Miura, T. Ishii, and H. Takamizawa, "Customizing virtual machine with fault injector by integrating with SpecC device model for a software testing environment D-cloud," in *Proc. IEEE 16th Pacific Rim Int. Symp. Dependable Comput.*, Dec. 2010, pp. 47–54, doi: 10.1109/PRDC.2010.37.
[47] H. Jitsumoto, Y. Kobayashi, A. Nomura, and S. Matsuoka, "MH-QEMU: Memory-state-aware fault injection platform," in *Supercomputing Frontiers*, D. Abramson and B. de Supinski, Eds. Cham, Switzerland: Springer, 2019.
[48] F. de Aguiar Geissler, F. Lima Kastensmidt, and J. Pereira Souza, "Soft error injection methodology based on QEMU software platform," in *Proc. IEEE LATW*, Mar. 2014, pp. 1–5.
[49] G. Di Guglielmo, D. Ferraretto, F. Fummi, and G. Pravadelli, "Efficient fault simulation through dynamic binary translation for dependability analysis of embedded software," in *Proc. 18th IEEE Eur. Test Symp. (ETS)*, May 2013, pp. 1–6.
[50] Y. Li, P. Xu, and H. Wan, "A fault injection system based on QEMU simulator and designed for BIT software testing," in *Proc. 2nd Int. Symp. Comput., Commun., Control Autom.*, 2013, pp. 580–587.
[51] J. An, H. You, F. Xie, Y. Yang, and J. Sun, "FIG-QEMU: A fault inject platform supporting full system simulation," in *Proc. 7th Int. Conf. Dependable Syst. Their Appl. (DSA)*, Nov. 2020, pp. 275–278, doi: 10.1109/DSA51864.2020.00049.
[52] N. Tian, D. Saab, and J. A. Abraham, "ESIFT: Efficient system for error injection," in *Proc. IEEE 24th Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Jul. 2018, pp. 201–206, doi: 10.1109/IOLTS.2018.8474160.
[53] R. Amarnath, S. N. Bhat, P. Munk, and E. Thaden, "A fault injection approach to evaluate soft-error dependability of system calls," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2018, pp. 71–76, doi: 10.1109/ISSREW.2018.00-28.
[54] A. Höller, G. Schönfelder, N. Kajtazovic, T. Rauter, and C. Kreiner, "FIES: A fault injection framework for the evaluation of self-tests for COTS-based safety-critical systems," in *Proc. 15th Int. Microprocessor Test Verification Workshop*, Dec. 2014, pp. 105–110, doi: 10.1109/MTV.2014.27.
[55] S. Chylek, "Emulation-based software reliability evaluation and optimization," *Przeglad Elektrotechniczny*, vol. 90, pp. 121–124, Feb. 2014, doi: 10.12915/pe.2014.02.32.
[56] S. Chylek and M. Goliszewski, "QEMU-based fault injection framework," *Studia Inform.*, vol. 33, pp. 25–42, Jan. 2012.
[57] M. Murciano and M. Violante, "Validating the dependability of embedded systems through fault injection by means of loadable kernel modules," in *Proc. IEEE Int. High Level Design Validation Test Workshop*, 2007, pp. 179–186.
[58] D. Ferraretto and G. Pravadelli, "Simulation-based fault injection with QEMU for speeding-up dependability analysis of embedded software," *J. Electron. Test.*, vol. 32, no. 1, pp. 43–57, Feb. 2016.
[59] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2005, p. 41.

[60] N. DeBardeleben, S. Blanchard, Q. Guan, Z. Zhang, and S. Fu, "Experimental framework for injecting logic errors in a virtual machine to profile applications for soft error resilience," in *Proc. Euro-Par Workshops*, 2011, pp. 282–291.
[61] Q. Guan, X. Hu, T. Grove, B. Fang, H. Jiang, H. Yin, and N. DeBadeleben, "Chaser: An enhanced fault injection tool for tracing soft errors in MPI applications," in *Proc. 50th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2020, pp. 355–363, doi: 10.1109/DSN48063.2020.00051.
[62] A. Davanian, Z. Qi, Y. Qu, and H. Yin, "DECAF++: Elastic whole-system dynamic taint analysis," in *Proc. 22nd Int. Symp. Res. Attacks, Intrusions Defenses (RAID)*, 2019, pp. 1–15.
[63] A. Höller, G. Macher, T. Rauter, J. Iber, and C. Kreiner, "A virtual fault injection framework for reliability-aware software development," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. Workshops*, Jun. 2015, pp. 69–74, doi: 10.1109/DSN-W.2015.16.
[64] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner, "QEMU-based fault injection for a system-level analysis of software countermeasures against fault attacks," in *Proc. Euromicro Conf. Digit. Syst. Design*, Aug. 2015, pp. 530–533, doi: 10.1109/DSD.2015.79.
[65] J. Xu and P. Xu, "The research of memory fault simulation and fault injection method for BIT software test," in *Proc. 2nd Int. Conf. Instrum., Meas., Comput., Commun. Control*, Dec. 2012, pp. 718–722, doi: 10.1109/IMCCC.2012.174.
[66] L. Wanner, S. Elmalaki, L. Lai, P. Gupta, and M. Srivastava, "VarEMU: An emulation testbed for variability-aware software," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Sep. 2013, pp. 1–10, doi: 10.1109/CODES-ISSS.2013.6659014.
[67] *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, document IEC/EN 61508, 2002.
[68] NASA. *NAS Parallel Benchmarks*. Accessed: Nov. 7, 2022. [Online]. Available: http://www.nas.nasa.gov/Software/NPB
[69] M. R. Guthaus and J. S. Ringenberg, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization (WWC)*, Dec. 2001, pp. 3–4.
[70] (2001). *DBench—Dependability Benchmarking (Project IST-2000-25425)*. [Online]. Available: http://www.laas.fr/DBench/
[71] S. Miura, T. Hanawa, T. Yonemoto, T. Boku, and M. Sato, "RI2N/DRV: Multi-link ethernet for high-bandwidth and fault-tolerant network on PC clusters," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–8.
[72] *RTEMS Application*. Accessed: Nov. 5, 2022. [Online]. Available: http://www.rtems.com
[73] S Artifact Infrastructure Repository. (2012). *Traffic-Collision-Avoidance System (TCAS)*. [Online]. Available: http://sir.unl.edu
[74] B. Branner, "The Mandelbrot set," in *Proc. Symp. Appl. Math.*, vol. 39, 1989, pp. 75–105.
[75] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Commun. ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984.
[76] *MPI Version of Matrix-Vector Product Computation*. Accessed: Dec. 2, 2022. [Online]. Available: https://people.sc.fsu.edu/~jburkardt/c_src/mpi/matvec_mpi.c
[77] *Cell-Based Adaptive Mesh Refinement*. [Online]. Available: https://github.com/losalamos/CLAMR
[78] *The Rodinia Benchmark Suite*. Accessed: Nov. 10, 2022. [Online]. Available: https://github.com/pathscale/rodinia/
[79] M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann, "dOSEK: The design and implementation of a dependability-oriented static embedded kernel," in *Proc. 21st IEEE Real-Time Embedded Technol. Appl. (RTAS)*, Los Alamitos, CA, USA: IEEE Computer Society Press, Apr. 2015, pp. 259–270.
[80] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2017, pp. 249–258, doi: 10.1109/ISPASS.2017.7975296.

YOHANNES B. BEKELE (Graduate Student Member, IEEE) received the B.S. degree in electrical engineering from Arba Minch University, Ethiopia, in 2007, and the M.S. degree in telecommunication networks engineering from Addis Ababa University, Ethiopia, in 2018. He is currently pursuing the Ph.D. degree with North Carolina Agricultural and Technical State University (NCA&T).

Since 2019, he has been a Graduate Research Assistant with the Automated Design for Emerging Processing Technologies (ADEPT) Laboratory, NCA&T, working on his research toward his Ph.D. dissertation. His research interests include microarchitectural and architectural reliability evaluation, hardware security and its relations with reliability, and microkernel evaluation methodologies.

DANIEL B. LIMBRICK (Senior Member, IEEE) received the B.S. degree in electrical engineering from Texas A&M and the M.S. and Ph.D. degrees from Vanderbilt University.

He is currently an Associate Professor with the Electrical and Computer Engineering Department, North Carolina Agricultural and Technical State University (NCA&T). He is a Postdoctoral Fellow with the Georgia Institute of Technology, he researched methods to improve routing congestion in monolithic 3D integrated circuits with Dr. Sung Kyu Lim. As a doctoral student, he was a member of the Radiation Effects and Reliability (RER) Group and the Security And Fault Tolerance (SAF-T) Research Group, where his research was conducted under the advisement of Dr. William H. Robinson. He leads the Automated Design for Emerging Processing Technologies (ADEPT) Laboratory, NC A&T, where he researches ways to improve the reliability and scalability of integrated circuits through logic and physical synthesis. His research interests include electronic design automation, post-CMOS technologies, computer architecture, lab-on-a-chip, and the reliability of microelectronics.

JOHN C. KELLY (Member, IEEE) received the Ph.D. degree in electrical engineering from the University of Delaware.

He is currently an Associate Professor with the Department of Electrical and Computer Engineering, North Carolina Agricultural and Technical State University (NCA&T). His research interests include hardware security in cyber-physical systems and embedded systems security. He contributes to research on engineering education, enhanced retention of underrepresented minorities in engineering, and hands-on learning techniques.