CONCURRENT, FLEXIBLE, AND PORTABLE FAULT INJECTION SYSTEM

By

YAO-CHIA CHUANG

Submitted in the partial fulfillment of the requirements

For the degree of Master of Science

Department of Electrical, Computer, and Systems Engineering

CASE WESTERN RESERVE UNIVERSITY

May, 2024

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

YAO-CHIA CHUANG

candidate for the degree of **Master of Science***.

Committee Chair

Dr. Daniel G. Saab

Committee Member

Dr. Christos Papachristou

Committee Member

Dr. Pan Li

Date of Defense

April 1, 2024

*We also certify that written approval has been obtained for

any proprietary material contained therein.

Table of Content

LIST OF TABLES

LIST OF FIGURES

CONCURRENT, FLEXIBLE, AND PORTABLE FAULT INJECTION SYSTEM

Abstract

By

YAO-CHIA CHUANG

The development of a Concurrent, Flexible, and Portable Fault Injection System represents a significant advancement in the field of system reliability and dependability testing. Leveraging the SWIFI (Software-Implemented Fault Injection), this thesis emphasizes the robust evaluation of software systems by introducing faults in a controlled manner. Notably, the system is tailored for applications written in C++ and makes extensive use of threading to simulate various fault scenarios concurrently. This approach enhances the efficiency of detecting potential system vulnerabilities and ensures a higher degree of flexibility and portability in testing procedures. By simulating real-world fault conditions, developers can identify and address vulnerabilities effectively, thereby improving the overall reliability and dependability of the system. This fault injection system stands out for its ability to provide comprehensive insights into system behavior under fault conditions, making it an invaluable tool for developers aiming to build resilient software applications.

Chapter 1: Introduction

Discussing fault injection and its nuances, particularly the assertion that no fault injection category is universally superior, requires an understanding of what fault injection is, its purposes, and the various categories it encompasses. This approach is crucial in recognizing that the efficacy and appropriateness of fault injection methods depend significantly on the context of the testing environment.

Fault injection is a testing technique used to validate a system's robustness and error-handling capabilities. By intentionally introducing faults or errors into a system, developers can observe how it behaves under unexpected conditions, ensuring it can handle such scenarios gracefully in a real-world environment. This technique is crucial for identifying and mitigating potential points of failure in both hardware and software systems, enhancing their reliability, security, and performance.

The essence of fault injection lies in its diversity; it encompasses various techniques, each suited to different systems and objectives. By injecting faults into a system and observing its behavior, researchers and practitioners can gain insights into its vulnerabilities and the efficacy of its fault tolerance mechanisms. This iterative process of injecting faults, analyzing system responses, and refining fault tolerance strategies forms the basis for achieving robust and dependable systems. Fault injection techniques can be broadly categorized into hardware-based, software-based, and simulation-based methods, each with unique advantages and application scenarios.

Hardware Based Fault Injection: it uses additional hardware to introduce faults into the target system's hardware. [1] Depending on the faults and their locations, hardware-implemented fault injection methods fall into two categories: with contact and without contact [1].

Software Based Fault Injection: it is also called as Software-implemented fault injection (SWIFI). It relies on the assumption that the effects of real hardware faults can be emulated either by manipulating the state of the target system registers and memory via run-time injection or by modifying the target workload through pre-run-time injection [2]. This assumption usually holds for transient faults, but for permanent faults, it presents some difficulty due to the repeated invocation of the fault injection exception handler every time a register or memory location is referenced [2].

Simulation Based Fault Injection: the target system and the possible hardware faults are modeled and simulated by a software program, usually called a fault simulator [3]. In this approach, the system or its environment is modeled in a simulator, and faults are introduced into the simulation. This method allows for exploring fault impacts in complex systems that are difficult or impractical to test in real life, such as satellite systems, without the risk of causing actual damage.

The choice among these fault injection categories should be guided by several factors, including the system's nature, the testing objectives, and available resources. For instance, hardware-based fault injection might be indispensable for testing embedded systems or hardware components where physical conditions can lead to failures.

Furthermore, the testing environment is critical in selecting the appropriate fault injection method. In a development environment, where the primary goal is identifying and fixing bugs early, software-based or simulation-based fault injection might be preferred due to its flexibility and safety. In contrast, hardware-based fault injection could be more appropriate in a staging or pre-production environment, where testing aims to mimic real-world conditions as closely as possible.

It's also essential to consider the system's criticality and the potential impact of failures. For high-stakes environments, such as in the aerospace, automotive, or healthcare industries, exhaustive testing using multiple fault injection techniques might be necessary to ensure the system's reliability and safety.

The fault injection tool we used in this thesis, ETFIDS, is a runtime-based software fault injection tool. It injects a fault by changing signal/variable values at runtime, and it also provides the ability to observe the effect on the output or behavior of the system [4].

Chapter 2: Fault Injection Techniques

As mentioned in Chapter 1, fault injection techniques can be broadly categorized into hardware-based, software-based, and simulation-based methods. There is no one-size-fits-all approach to fault injection. The effectiveness and appropriateness of a fault injection category are highly contingent on the specific context of the testing environment, the nature of the system under test, and the objectives of the testing process. A nuanced understanding of these factors is crucial for selecting the most suitable fault injection method. Developers and testers must carefully evaluate their options, considering the trade-offs between realism, risk, cost, and the comprehensive assessment of the system's fault tolerance capabilities. We will talk about a few notable examples for each category.

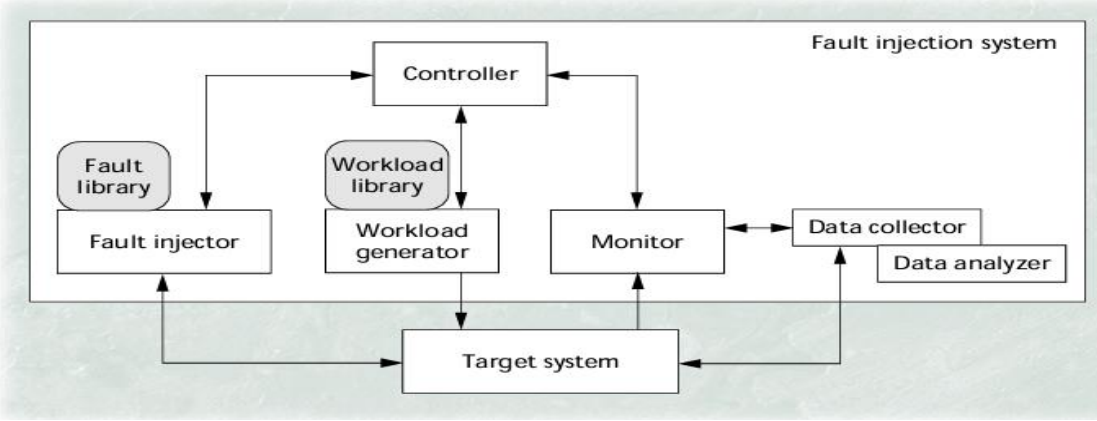

Figure **1**: Basic components of a fault injection environment. [1]

2.1 Hardware Based Fault Injection

2.1.1 Radiation Testing in Spacecraft Systems:

Radiation testing in spacecraft systems is critical to ensuring the reliability, performance, and longevity of spacecraft operating in harsh space environments. Exposure to various forms of radiation in space, including cosmic rays, solar flares, and Van Allen belt radiation, can cause significant damage to onboard systems and components. This damage can range from temporary malfunctions to permanent failures, potentially compromising mission objectives and safety. Therefore, understanding and mitigating radiation effects through rigorous testing is paramount for the success of space missions.

There are three types of space radiation–particles inside Earth's magnetic field, particles shot through space by solar Particle events (such as solar flares), and the heavy ions and high-energy protons produced outside this solar system (galactic cosmic rays) [5]. The energy levels of these particles can be extremely high, capable of penetrating spacecraft shielding and causing direct ionization and displacement damage to materials and electronic devices.

Radiation can affect spacecraft systems in several ways. Single Event Effects (SEEs), which can be either destructive or nondestructive, occur when a single energetic particle, such as high-energy heavy ions or protons, passes through a semiconductive material, depositing energy and potentially increasing the risk of failure on a space mission [6]. There are four types of SEEs: Single Event Burnout(SEB), Single Event Upset (SEU), Single Event Transients (SETs), and Single Event Functional Interrupts (SEFIs). They can result in minor issues and catastrophic failure. Total Ionizing Dose (TID) significantly affects electronic devices in space, where prolonged exposure to ionizing radiation can increase transistor leakage currents and alter threshold voltages and sub-threshold slopes, potentially leading to component failure or complete malfunction if sufficiently high [7].

This testing is crucial for assessing a component's vulnerability to TID radiation effects, aiming to identify parameter variations across the total dose and the threshold where performance falls short of mission requirements while also highlighting that radiation effects on space electronics pose risks to technology manufacturers, as failure or malfunction due to TID can endanger missions and lives, underscoring the importance of thorough pre-mission evaluation to mitigate TID radiation impact. Displacement damage caused by non-ionizing energy loss can alter the physical structure of materials, affecting their mechanical and electrical properties.

To assess and mitigate these risks, spacecraft systems undergo comprehensive radiation testing during their development phase. Radiation testing involves exposing spacecraft components and systems to controlled radiation sources that simulate the space environment. This testing can be conducted at various levels, including component, subsystem, and system levels, to identify vulnerabilities and validate protective measures.

Component-level testing, subsystem, and system-level testing, along with simulation and modeling, form the backbone of ensuring spacecraft resilience against the harsh radiation environment encountered in space. This comprehensive approach ensures that individual components and entire systems can withstand the various forms of radiation they will be exposed to, thus safeguarding the mission's integrity and the safety of any crew.

At the heart of space mission durability against radiation lies component-level testing. This meticulous process involves evaluating individual electronic parts under controlled radiation exposure to assess their vulnerability to Single Event Effects (SEEs),

Total Ionizing Dose (TID), and displacement damage. SEEs occur when a single energetic particle, such as a proton or heavy ion, strikes a semiconductor, leading to temporary or permanent damage. TID refers to the cumulative damage caused by exposure to ionizing radiation over time, which can degrade the performance of electronic components. Displacement damage results from particles displacing atoms in the material's lattice, potentially altering its physical properties and electronic behavior.

The primary goal of component-level testing is twofold: first, to identify each part's radiation sensitivity, and second, to determine its suitability for space applications. By understanding the limitations and thresholds of individual components, engineers can make informed decisions regarding selecting radiation-hardened (designed to withstand radiation effects) or radiation-tolerant (able to operate under a certain level of radiation) components. This strategic selection is crucial for developing spacecraft capable of enduring the rigors of space without succumbing to radiation-induced failures.

Moving beyond individual components, subsystems, and system-level testing provides a broader perspective on a spacecraft's resilience to radiation. In this phase, clusters of components are assembled into subsystems, and the complete spacecraft systems undergo rigorous radiation exposure tests. The purpose is to evaluate the collective response of these assemblies to space radiation, simulating as closely as possible the actual conditions they will face.

This level of testing is invaluable for several reasons. First, it helps assess the effectiveness of shielding solutions designed to protect sensitive electronics from radiation. Second, it allows for the evaluation of error correction codes and redundancy

strategies implemented within the hardware and software to mitigate the impact of radiation-induced errors. Lastly, system-level testing offers insights into the overall robustness of the spacecraft, identifying potential weak links in the design that could compromise mission success.

Parallel to physical testing, simulation and modeling are indispensable tools in understanding and mitigating radiation effects on spacecraft systems. Using advanced computational models and simulations, engineers can predict the interaction between radiation and electronic components, evaluate the potential impact of different space radiation environments, and test the efficacy of shielding and other protective measures.

These simulations can be incredibly detailed, encompassing various aspects of radiation physics, materials science, and electronic circuit behavior. They allow for exploring numerous scenarios, including extreme events that are difficult or impossible to recreate in a laboratory setting. By integrating simulation results with empirical data from physical testing, engineers can refine their designs, optimize radiation protection strategies, and enhance the reliability and safety of space missions.

The combined efforts in component-level testing, subsystem and system-level testing, and simulation and modeling are essential to developing spacecraft capable of surviving and functioning in a hostile space environment. This multi-layered approach enables engineers to identify vulnerabilities early in the design process, implement effective mitigation strategies, and ensure the success and safety of space missions. As humanity's space exploration expands, the importance of comprehensive radiation testing

and modeling will only grow, underpinning the reliability and longevity of future space endeavors.

2.1.2 Pin-Level Fault Injection on Microprocessors:

[8] discusses the architecture of a pin-level fault injector named RIFLE, designed for dependability validation of computing systems. RIFLE can adapt to various target systems, injecting faults primarily into the processor pins. The system offers deterministic fault injection with reproducibility without requiring feedback circuits to detect error production. It can identify instances where faults do not impact the target system, generating sets of faults that specifically affect the system's operations. The results show significant error detection with simple mechanisms, suggesting that computers with basic error detection approaches are closer to achieving fail-silent operation.

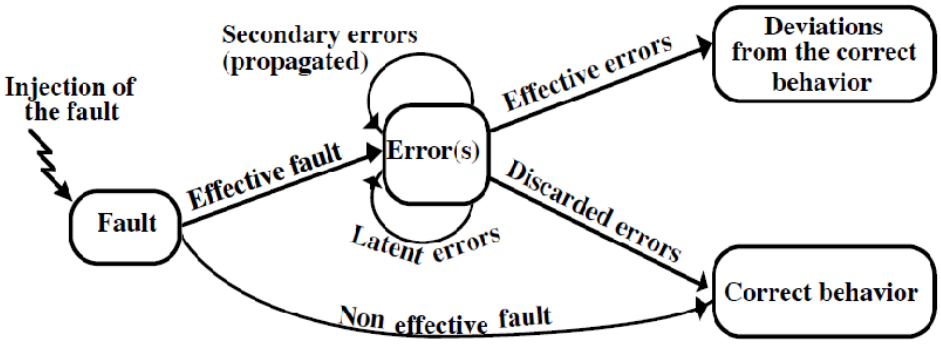

Figure **2**: Impact of a pin-level fault in the target system at the processor instruction level behavior [8]

Validating fault-tolerance mechanisms is challenging due to the complexity involved in fault activation and error propagation processes. [8] introduces RIFLE, a system that complements traditional validation techniques like modeling and simulation. RIFLE focuses on injecting physical faults into actual systems, highlighting the influence of workloads on error-handling mechanisms' performance.

RIFLE's architecture is detailed, showcasing its ability to inject faults at the pin level across various components of a computing system, mainly focusing on the processor. Faults can be triggered under specific conditions, allowing for controlled and reproducible fault injection. The system's adaptability to different target systems and capacity to detect the practical impact of injected faults without external feedback mechanisms are emphasized.

An evaluation of simple, behavior-based error detection techniques reveals that up to 72.5% of errors can be detected with basic mechanisms. [8] discusses the impact of different fault sets on detection coverage and latency, providing insights into the effectiveness of built-in error detection mechanisms in processors like the 68000 built-in error detection mechanisms. It also evaluates memory access error detection mechanisms, watchdog timers, and the fail-silent behavior in systems equipped with these error detection techniques.

The fail-silent behavior of systems employing behavior-based error detection mechanisms is scrutinized. The study reveals that over 90% of the faults led to systems behaving according to the fail-silent model. This indicates that traditional computers with simple error detection mechanisms are relatively close to achieving fail-silent operation,

highlighting the potential for enhancing system dependability through basic error detection strategies.

RIFLE presents a novel approach to fault injection for system dependability validation, showcasing its versatility and effectiveness. The system's capacity for deterministic fault injection and its ability to generate specific fault sets and assess their impact without external feedback mechanisms marks a significant advancement in fault injection methodologies. The evaluation of simple error detection mechanisms underscores the potential of basic strategies in achieving fail-silent operation, paving the way for more dependable computing systems.

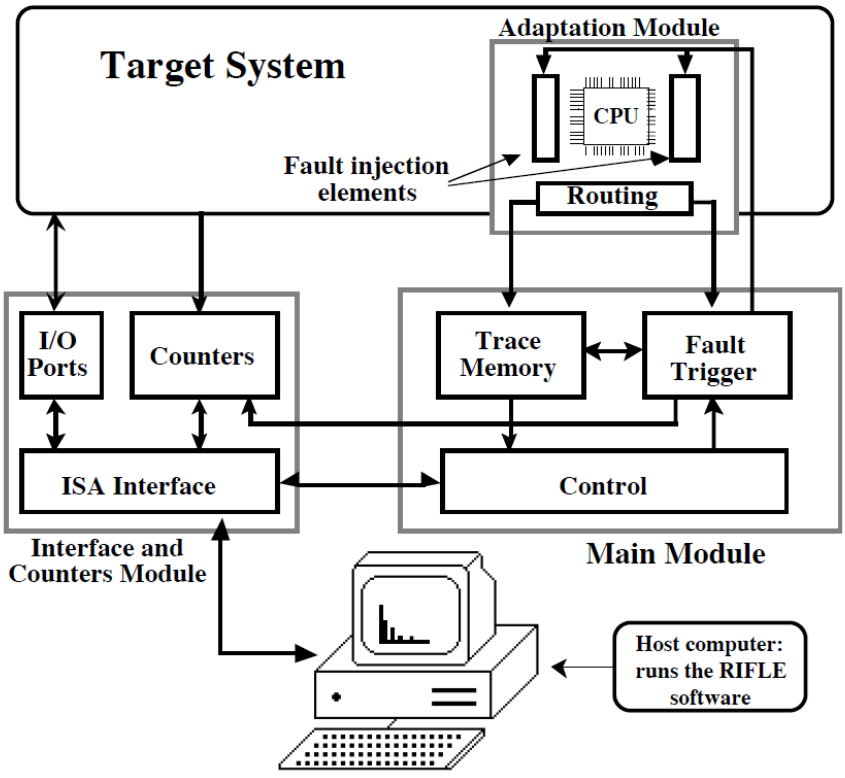

Figure **3**: RIFLE organization. [8]

2.2 Software Based Fault Injection

2.2.1 Chaos Monkey in Cloud Computing Environments:

Thirty years ago, Jim Gray noted that "A way to improve availability is to install proven hardware and software, and then leave it alone" [9] [10]. In the recent era that demands putting customers first and emphasizes after-sales service, it's impossible for companies to “leave it alone” after it's sold. Companies must ensure the product can function correctly when selling or providing a service. Today's business giants like Google [11], Facebook [12], Microsoft [13], and Amazon [11] all have corresponding measures to ensure their services operate smoothly. As a streaming platform with tens of millions of subscribers, Netflix also has measures known as "Chaos Monkey." After all, no one wants their relaxation time to be ruined by the unreliability of a streaming platform.

For years, Netflix has been running an internal service called Chaos Monkey [14], which randomly selects virtual machine instances that host our production services and terminates them [10]. Chaos Monkey's purpose was to encourage Netflix engineers to design software services that can withstand failures of individual instances [10]. The inception of Chaos Engineering can be traced back to the acknowledgment of the inherent complexity and failure modes of distributed systems. Traditional engineering practices, which might suffice in more monolithic or less dynamic environments, fail to address the challenges posed by modern, distributed internet-scale services. The critical insight was that, despite best efforts in design and testing, unforeseen failures are

inevitable in complex systems. This realization prompted a shift in focus from merely trying to prevent all possible failures to ensuring that systems are resilient and can maintain functionality in the face of disruptions. Chaos Engineering has four principles: building hypotheses around steady-state behavior, varying real-world events, running experiments in production, and automating experiences.

The cornerstone of Chaos Engineering is identifying and understanding a system's steady state, which represents its normal operating conditions. At Netflix, metrics such as "stream starts per second" (SPS) serve as indicators of this steady state, providing a measurable and observable output that reflects the health and availability of the service. Experiments in Chaos Engineering are designed around perturbing this steady state in controlled ways to test hypotheses about the system's resilience.

Chaos Engineering involves introducing changes that simulate real-world events, ranging from server crashes and network partitions to more subtle conditions like increased latency or load. Historical incidents and theoretical analysis of potential failure modes inform the selection of these events. This principle emphasizes the importance of testing the system's response to various stressors rather than limiting scrutiny to those failures already experienced or most easily imagined.

One of the more controversial aspects of Chaos Engineering is the insistence on running experiments in the actual production environment. This approach stems from the understanding that distributed systems' complexity and emergent behavior can never be fully replicated in a test environment. Running experiments in production ensures that

findings are as relevant and accurate as possible, directly informing improvements in system resilience.

Chaos Engineering experiments must be automated and run continuously to remain effective as systems evolve. This automation allows for consistency, and The system's resilience is continually validated against a backdrop of constant change—new code deployments, configuration changes, and evolving user behaviors. Automation also facilitates the scaling of Chaos Engineering practices, enabling them to cover more aspects of the system and adapt dynamically to new insights and conditions.

Netflix's adoption of Chaos Engineering showcases a comprehensive approach to resilience, encompassing everything from individual service robustness to the integrity of its content delivery network. Tools like Chaos Monkey and Chaos Kong have become emblematic of this approach, each targeting different levels of the system's architecture to ensure that every layer is prepared to handle failures gracefully.

Beyond Netflix, the principles of Chaos Engineering have begun to influence a wide range of organizations and systems. As digital services become increasingly central to all aspects of modern life, maintaining availability and function in the face of unexpected disruptions has become critical. Chaos Engineering provides a framework and methodology for achieving this resilience, grounded in empirical testing and continuous improvement.

Chaos Engineering represents a paradigm shift in approaching system reliability and resilience. By embracing failure as a means to learn and improve, organizations can build systems that are not only capable of surviving unexpected disruptions but are also

more robust, flexible, and responsive to the demands of the digital age. The principles and practices outlined in Netflix's document offer valuable insights and a roadmap for implementing Chaos Engineering across various contexts, promising to play a pivotal role in the evolution of technology infrastructure.

2.2.2 LLFI (LLVM Fault Injection Tool):

[15] introduces LLFI (Low-Level Fault Injection), an innovative fault injection tool designed to evaluate software resilience against hardware faults. As hardware errors become increasingly prevalent due to reducing feature sizes in microelectronic devices, ensuring software resilience against these errors has become a pivotal challenge. Traditional hardware-centric solutions for error resilience are becoming cost-prohibitive, pushing the research toward software-based error resilience strategies. LLFI represents a significant advancement in this research area by enabling precise, configurable fault injections at the LLVM (Low-Level Virtual Machine) Intermediate Representation (IR) level.

[15] presents LLFI as a software-implemented fault injection (SWiFI) tool that operates at the LLVM IR level, bridging the gap between high-level source code analysis and low-level hardware operation emulation. This allows for accurate and configurable fault injections, making it a versatile tool for researching and enhancing software error resilience techniques. Through extensive experiments involving nine benchmark programs, [15] demonstrates LLFI's utility in investigating the impact of various fault injection parameters (such as instruction type, register target, and bit flip count) on application resilience. The experiments reveal significant insights into how these

parameters influence the failure modes of applications, thereby guiding the development of more resilient software systems.

The findings from LLFI's application suggest that instruction type significantly influences failure outcomes, injecting faults into source registers typically leads to higher crash rates than into destination registers, and the distinction between single and double-bit flips has minimal impact on Silent Data Corruption (SDC) rates. These insights underscore the nuanced nature of software error resilience and the importance of targeted fault injection strategies in evaluating and enhancing resilience. LLFI leverages the LLVM compiler infrastructure to inject faults into selected program points in a fine-grained manner. It consists of two main components: LLVM passes for static code analysis and instrumentation and runtime libraries for executing the fault injections based

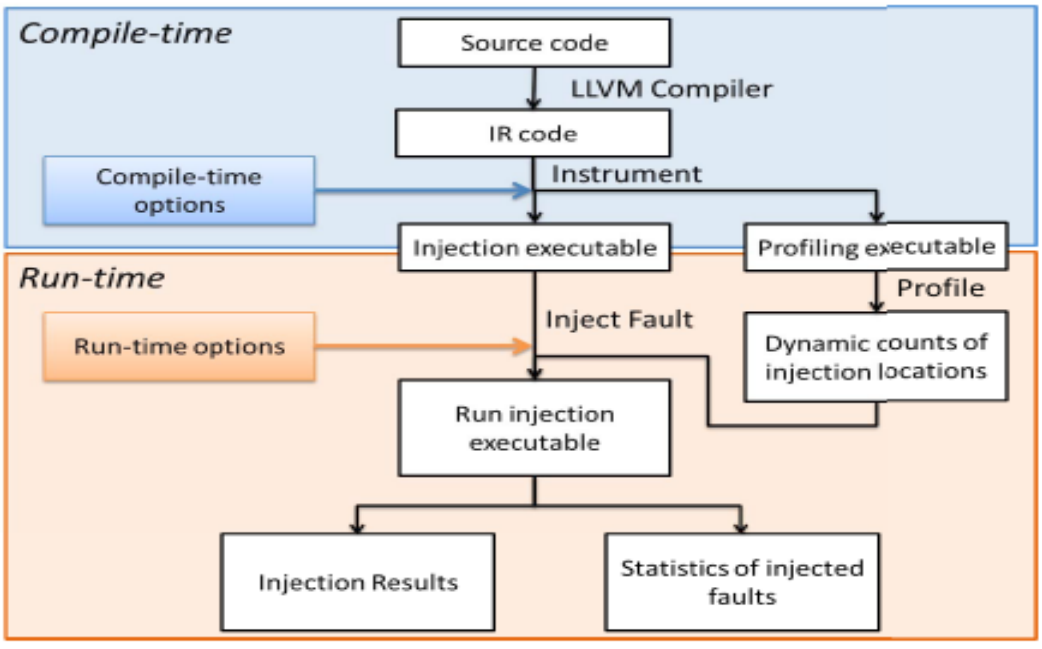

Figure **4**: LLFI workflow from the user perspective. [15]

on user-defined parameters. This design enables LLFI to support various programs and programming languages, making it a flexible tool for resilience studies.

[15] not only present their results but also delve into their significance, exploring how differences in instruction types, register targets, and fault types can influence application failure modes. This analysis is grounded in the data obtained through LLFI, highlighting the tool's practical utility in advancing our understanding of software resilience. Moreover, it situates LLFI within the broader context of fault injection research, offering a review of related work encompassing program-level and assembly code-level fault injection techniques. This review underscores LLFI's novelty and its contribution to the field.

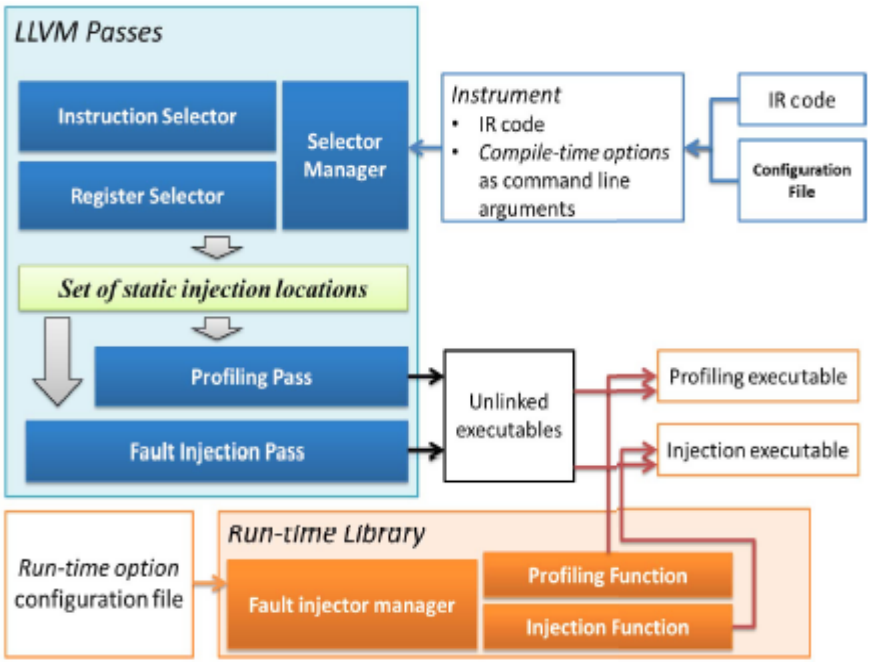

Figure **5**: LLFI workflow from the user perspective. [15]

In conclusion, [15] articulates a compelling case for LLFI as a powerful tool for researching software error resilience. By enabling precise, configurable fault injections at the LLVM IR level, LLFI opens up new avenues for understanding and improving the strength of software systems against hardware faults. The authors' thorough experimental evaluation and insightful analysis of the results significantly contribute to the ongoing effort to develop more robust and error-resilient software systems.

2.2.3 IM-FIT:

IM-FIT [16] has been introduced as a versatile tool designed to evaluate software robustness in safety-critical systems, focusing on Python-based and ROS-based systems. Its development aims to address the need for rigorous testing methodologies that can simulate a wide range of fault conditions to ensure system reliability and safety.

The significance of IM-FIT lies in its application to safety-critical systems, where failure can result in significant harm or loss. The tool's mutation-based testing approach represents a proactive strategy to identify and mitigate potential system failures. IM-FIT's contribution to the field is underscored by its capacity to generate comprehensive fault libraries and implement mutation-based testing methods, which are crucial for critical systems' verification and validation (V&V).

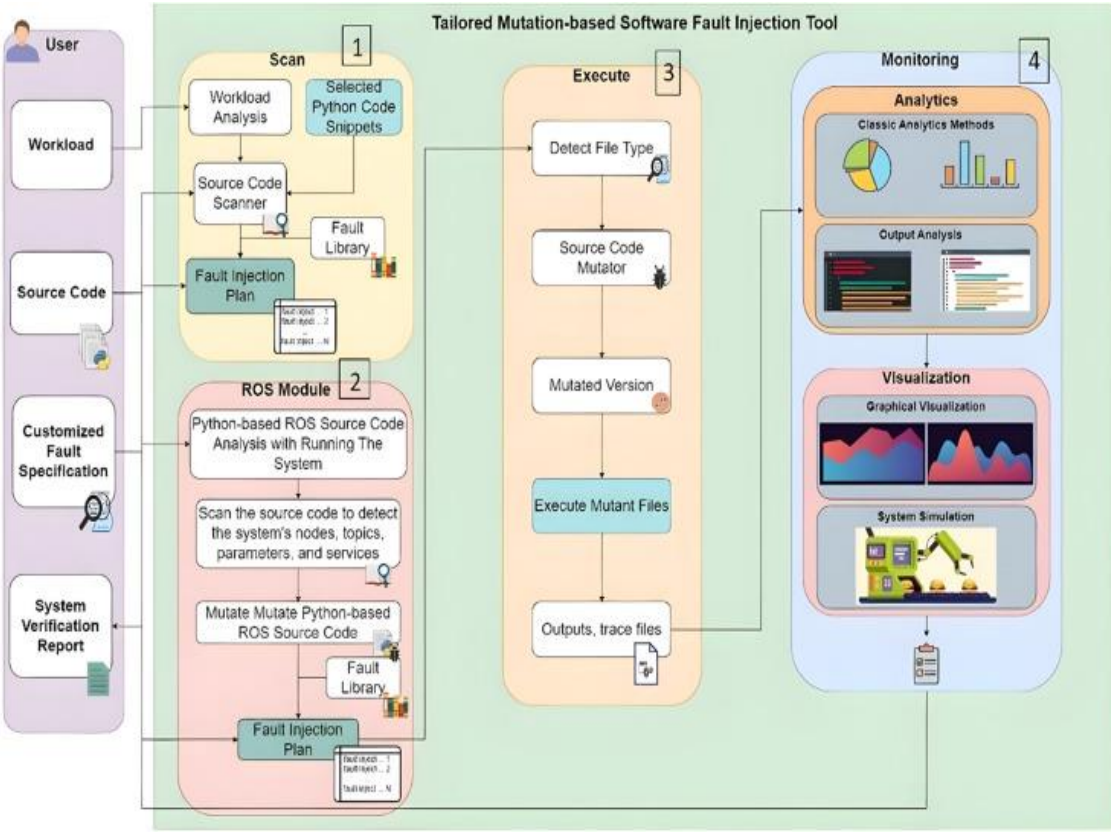

Figure **6**: IM-FIT usage architecture. [16]

The architecture of IM-FIT is detailed, illustrating its components and functionality. The software's design allows for identifying fault-applicable lines in source code, leveraging a customizable RegEx and AST-based structure for fault injection. Key functionalities are described, including the evaluation of Python-based software robustness and the ROS mutation module, which underline the tool's adaptability and specificity in testing different system types.

IM-FIT utilizes mutation-based testing, where artificial faults are injected into the system to assess its response and adaptability. This method provides insights into potential weaknesses and areas for improvement. The software's execution metrics are introduced to evaluate software robustness. These include Detected Mutations,

Undetected Mutations, Valid Mutations, Invalid Mutations, Total Mutations, and the Mutation Score, which collectively offer a comprehensive overview of the system's resilience.

Examples of IM-FIT's application demonstrate its utility in scanning launch files, extracting critical information, and injecting faults into ROS-based systems. These examples highlight the practical benefits of using IM-FIT in real-world scenarios. The impact of IM-FIT extends beyond its technical capabilities, offering significant benefits in terms of time, effort, and cost savings in the V&V process. Its unique features, particularly for ROS-based testing, underscore its value in managing complex projects and ensuring system robustness. IM-FIT's role in facilitating the rapid completion of safety-critical system studies, generating datasets for AI training, and aiding in quality certification processes is discussed.

2.3 Simulation Based Fault Injection

Simics

Simics is a full system simulator developed by Virtutech AB, designed to strike a balance between accuracy and performance in system simulation. It aims to model complete final applications and provide a unified framework for both hardware and software design. The importance of simulation in computer architecture design is well-established, dating back to early projects like the EDSAC in the 1950s. Simics builds on the principle that all computers can simulate each other, a concept stemming from the theoretical work of Alan Turing and Alonzo Church.

The complexity of modern digital systems necessitates the design and testing of hardware and software within the context of their final application. Traditional simulation methods often fall short by focusing on overly simplified models or "toy" workloads, leading to accurate but irrelevant results. Simics addresses this by offering a platform capable of running commercial workloads and interfacing with detailed hardware models, thereby providing both functional and timing accuracy.

Simics is designed to be sufficiently detailed to run unmodified operating systems and applications, making it versatile for simulating various system types, from embedded devices to high-end servers. It supports multiple processor architectures and operating systems, offering the flexibility to model complex, heterogeneous networks. Notably, Simics enables the simulation of intricate setups, such as telecom switches,

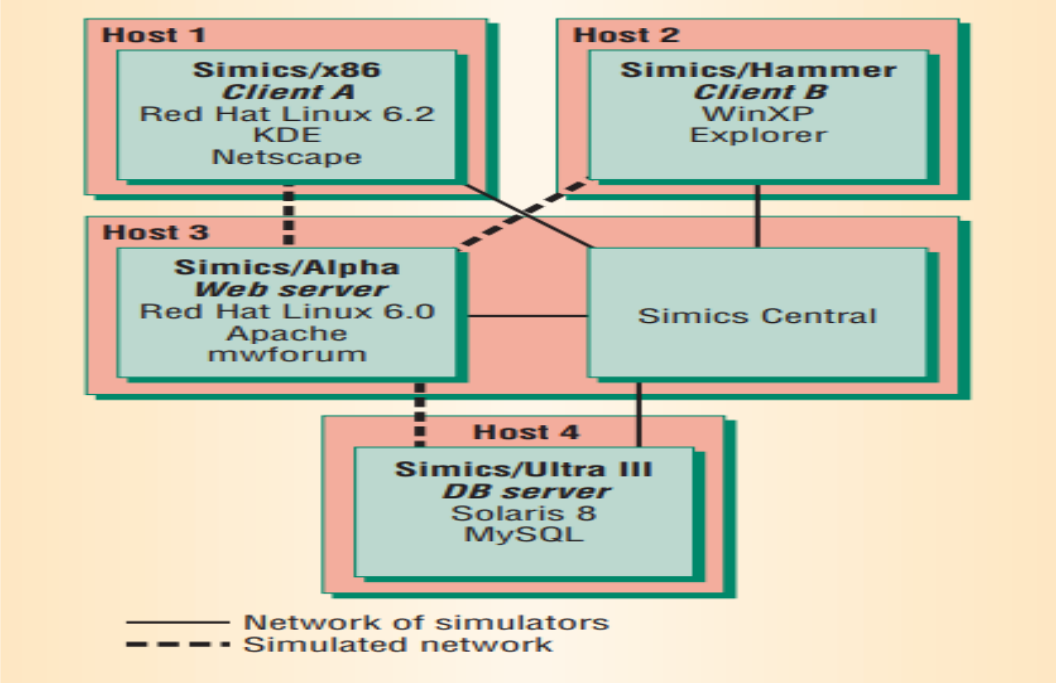

Figure 7: Sample network setup for Simics simulation. [17]

multiprocessor systems, and clusters, with the ability to run realistic workloads like the SPEC CPU2000 benchmark suite and database benchmarks.

Simics architecture includes a core module that provides basic simulation features such as processor instruction set and memory simulation. An extensive application programming interface (API) allows for the addition of specific device models and intrinsic components, enhancing Simics' extensibility. The system uses a simple object-oriented configuration language for system description, enabling easy modification and extension of the simulation environment.

Simics is utilized across various stages of system development, including microprocessor design, memory studies, device development, operating system development, debugging, and high-availability testing. Its performance is benchmarked across different processor architectures, demonstrating its capability to simulate complex systems efficiently.

Simics builds upon previous efforts in system simulation, such as IBM's early emulator and academic projects like SimOS. However, it distinguishes itself by running completely unmodified kernel and driver code across a heterogeneous network of systems. Simics represents a significant advancement in the field of system simulation, offering a comprehensive platform that supports a wide range of applications in computer architecture and system design.

The development and deployment of Simics have significant implications for the design, testing, and implementation of digital systems. By providing a platform that can accurately simulate complete systems and run realistic workloads, Simics facilitates a

more efficient and effective approach to system development. Its flexibility and extensibility make it a valuable tool for exploring new architectures, testing software in a controlled environment, and debugging complex systems. Looking forward, Simics has the potential to shape the future of system simulation, offering a foundation for the development of more sophisticated and accurate simulation methodologies.

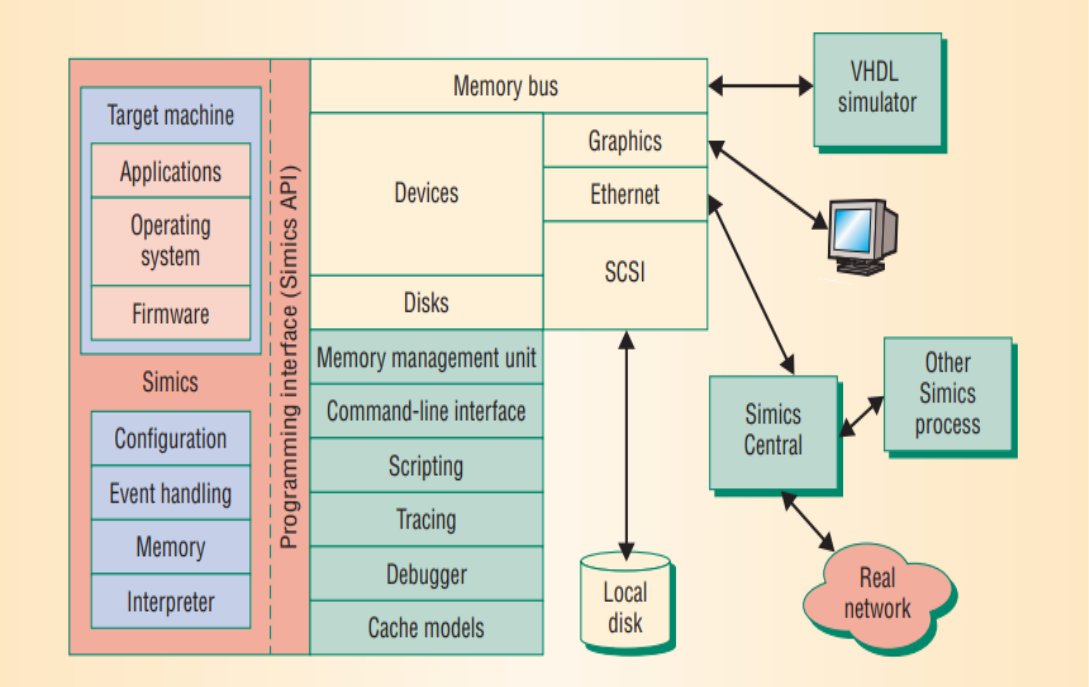

Figure **8**: Simic architecture. [17]

[17]presents a comprehensive overview of Simics, a full system simulator that balances accuracy and performance. By enabling detailed simulation of complete systems and supporting a wide range of applications, Simics represents a significant advancement in the field of system simulation. Its impact extends across the electronics industry, offering a powerful tool for system design, development, and testing. As digital systems continue to evolve in complexity, platforms like Simics will play a crucial role in facilitating innovation and ensuring the reliability and performance of future technologies.

Chapter 3: Enhancing Multithreaded Application Dependability with ETFIDS

3.1 Motivation

The development of the efficient transient fault injection and detection system (ETFIDS) [4] marks a significant advancement in addressing the specific challenges of conducting fault injection tests in multi-threaded environments. The complexity inherent in these environments, characterized by concurrent execution and intricate synchronization mechanisms, demands a fault injection solution beyond traditional tools' capabilities. ETFIDS stands out by offering this solution, directly responding to the need for precise, reproducible, and efficient testing methodologies tailored for the nuanced dynamics of multi-threaded applications.

Multi-threaded environments pose unique challenges for fault injection due to their complex behavior patterns and the potential for unpredictable interactions between threads. Traditional software fault injection tools often fall short in these scenarios, primarily due to their inability to accurately target faults and assess their impact in real-time across multiple threads. This limitation not only makes it difficult to simulate specific fault scenarios but also hinders the analysis of fault propagation and the system's resilience to errors.

ETFIDS addresses these challenges head-on by enabling precise fault specification and concurrent evaluation of system behaviors under both faulty and fault-

free conditions. This capability is crucial for multi-threaded environments, where the timing and location of a fault can significantly influence the system's overall behavior and stability. By providing a mechanism for detailed fault modeling and real-time error detection, ETFIDS facilitates a deeper understanding of fault tolerance and system dependability within the complex multi-threading context.

Furthermore, ETFIDS's integration with GDB for dependability analysis enhances its utility in multi-threaded applications, allowing for a comprehensive and nuanced analysis of faults and their effects. This integration ensures that ETFIDS can effectively navigate the complexities of multi-threaded systems, offering accurate and actionable insights.

The motivation behind employing ETFIDS in multi-threaded environments is its innovative approach to fault injection and detection. By addressing the unique challenges presented by these environments, ETFIDS not only improves the reliability and robustness of software systems but also pushes the boundaries of what is possible in software fault injection research. Its development represents a tailored response to the intricate requirements of multi-threaded testing, ensuring that researchers and developers can conduct more effective, efficient, and meaningful fault injection experiments.

3.2 The Overall Approach

ETFIDS is designed to conduct fault injection trials within a running application to determine how faults impact system behavior. In these experiments, depicted in Figure 9, ETFIDS introduces errors into the application while it is in operation, according to predefined user parameters. It then proceeds to monitor the application, observing how

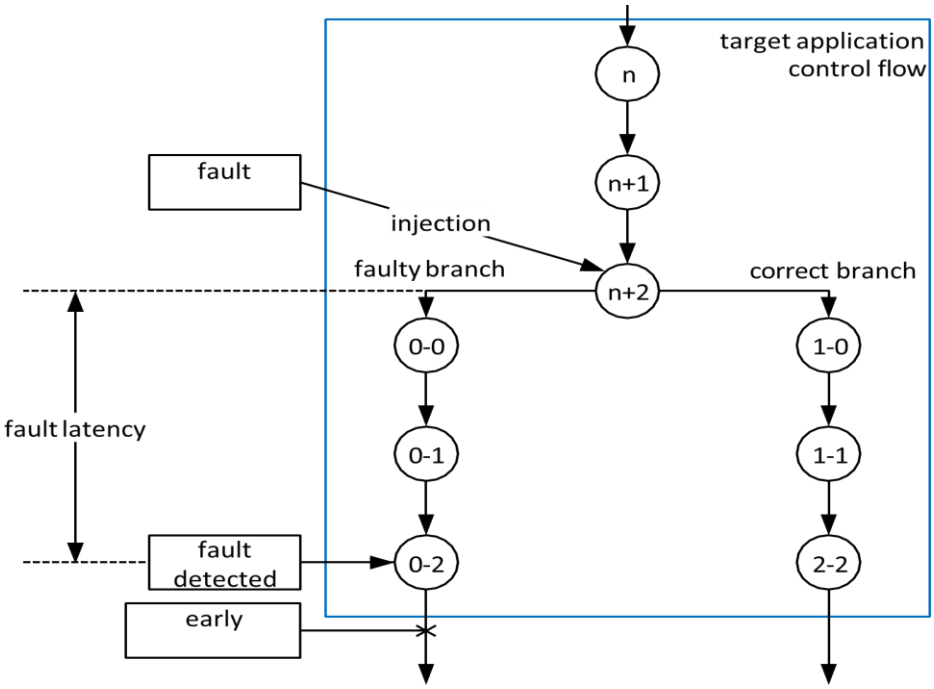

Figure **9**: Illustration of ETFIDS Fault Injection and Analysis. [4]

the errors propagate in real-time. When an error is recognized, ETFIDS records the time elapsed since the fault introduction—known as fault latency—and other pertinent data, after which it shuts down the application. This termination is intentional, as the user does not require further analysis post-error detection, thus optimizing the overall duration of fault injection testing.

Figure 10 delineates the comprehensive architecture of a fault injection framework that integrates a target application with the ETFIDS and the GNU Debugger (GDB). This figure represents the complex interplay between the user, ETFIDS, and the target application, demonstrating the fault injection and monitoring process.

The system hinges on the target application, which the user operates, providing ETFIDS with a specification file containing a detailed list of potential faults and points of

observation. This file is crucial for ETFIDS to configure the fault injection parameters. Once the fault injection experiment commences, ETFIDS leverages GDB, which uses the ptrace system call to control the application at runtime. GDB is pivotal in the fault injection process—it sets breakpoints and watchpoints based on the user's specified fault list and can manipulate the application's memory contents to induce faults.

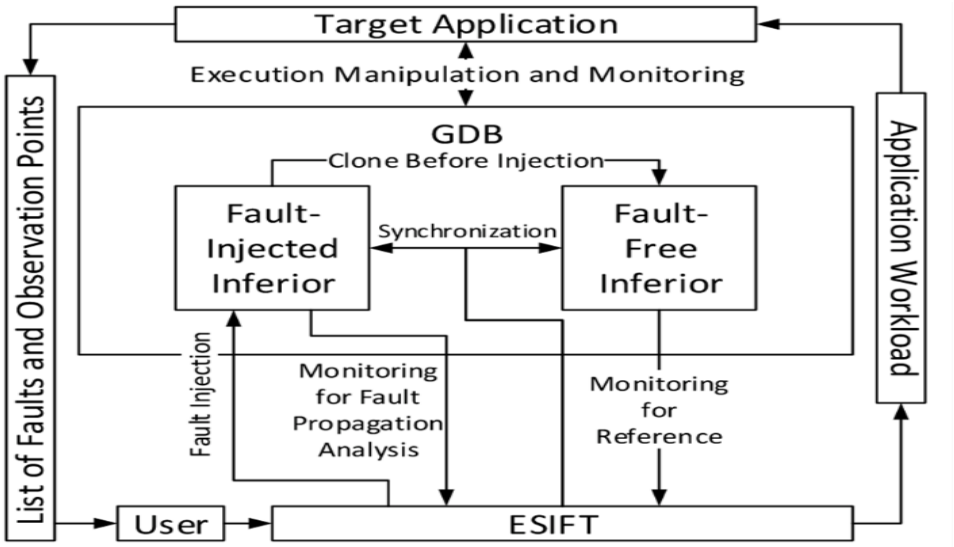

Figure **10**: ESIFT structure [4].

ETFIDS employs GDB to create a clone of the application's process before fault injection. It maintains two parallel executions of the application within GDB, termed "inferiors": one that carries the injected fault ("Fault-Injected Inferior") and one that runs faultlessly ("Fault-Free Inferior"). ETFIDS meticulously synchronizes these inferiors to ensure that the comparison for error propagation analysis is consistent and accurate. When errors arise from the injected faults, ETFIDS quickly identifies them by comparing the state of both inferiors at the designated observation points.

The user retains control over the entire fault injection process. They provide ETFIDS with a list specifying each fault, which includes the targeted variable's name, the code location for the fault injection, and the precise execution event triggering the fault. This specificity ensures accurate fault injection timing, uninfluenced by disparate testing environments. ETFIDS presumes visibility of all variables within the target system, and it assumes the user can query these via GDB, which is particularly accessible if the target application is compiled with the debug flag—this generates a symbol table that ETFIDS relies upon for identifying variable and function names.

Additionally, ETFIDS exhibits the capacity for fault outcome analysis through the use of two inferiors in GDB. GDB represents each program's execution state with an entity known as an inferior, which typically correlates to a process but can represent other types of program executions. ETFIDS maintains synchronization between the inferiors by establishing additional watchpoints, enabling the detection of discrepancies between the fault-injected and fault-free observations.

To optimize efficiency and conserve resources, ETFIDS is programmed to clone the target application's process precisely before the fault injection occurs. It utilizes hardware breakpoints offered by GDB, allowing ETFIDS to set breakpoints directly at the hardware level without altering the application's code. Most modern microprocessors support these hardware breakpoints and allow for less intrusive monitoring of target variables, reducing CPU overhead and accelerating the fault injection software's performance.

In essence, Figure 9 provides an intricate view of how ETFIDS functions, showcasing its dual capabilities in fault injection and analysis by employing GDB for real-time manipulation and monitoring of a target application's execution, facilitating a controlled environment for dependability assessment.

In a live system, simulating transient faults is essential to evaluate the resilience of the system at a circuit level. Research has demonstrated that these faults often cause a reversal in the state of a memory element's stored bit—a phenomenon widely known as a bit-flip. Figure 10 illustrates the conditions under which a bit-flip fault might occur during the operation of an application.

The diagram in Figure 10 highlights how environmental factors, such as particle strikes, can lead to transient faults. When a high-energy particle impacts a circuit node, it induces a pulse, often called a "single-event transient (SET) fault." Initially, this pulse does not alter the application's functioning. However, as the pulse moves along the circuitry and arrives at a memory element, it can cause a bit-flip by inverting the stored bit's value. If this pulse interacts with a counter, the fault might result in the accidental increment or decrement of the stored value, leading to a deviation in the application's expected behavior.

ETFIDS is equipped to simulate various transient fault models to analyze their impacts thoroughly. The current suite of transient fault models supported by ETFIDS includes:

1. Single bit-flip: A solitary bit in a memory element changes its state.

2. Multiple bit-flips: Several bits within a memory element simultaneously flip their states.

3. Increment: A value within a memory element is erroneously increased.

4. Decrement: A value within a memory element is erroneously decreased.

5. Force value: A memory element is coerced into adopting a specific value, which can occur due to significant crosstalk effects in densely packed circuits.

6. Jump: This fault can be induced externally, often by a cyber-attack, compelling the application to execute or branch to a set of instructions dictated by the attacker.

Such "force value" faults result when the circuit layout's compactness amplifies the crosstalk effect, influencing adjacent gates and causing a memory element to take on a specific value as the fault propagates. On the other hand, "jump" faults are typically

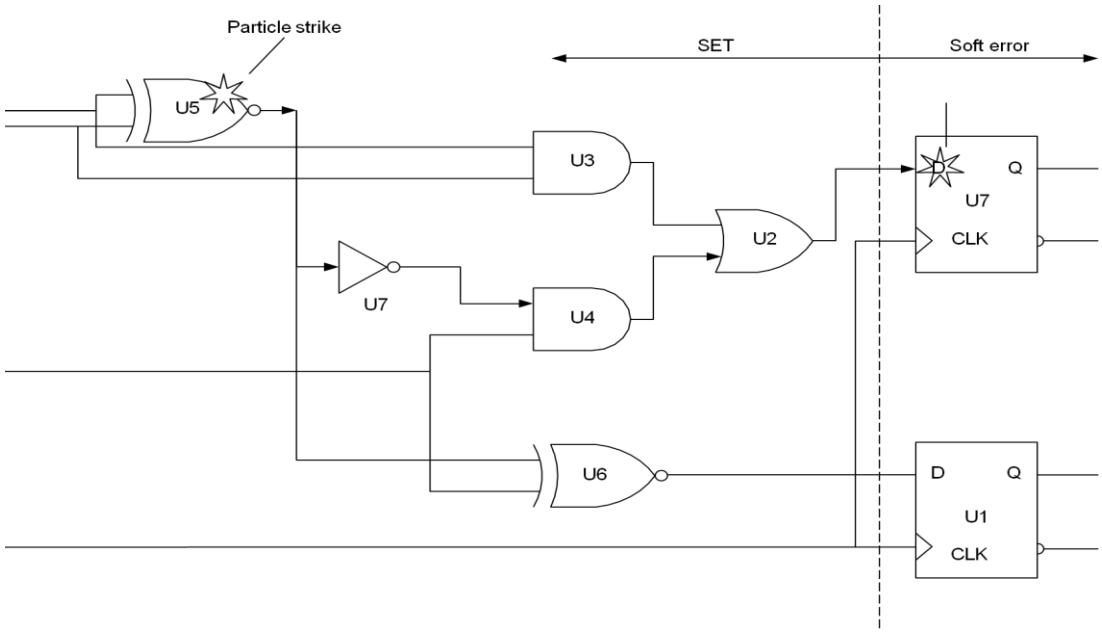

Figure **11**: Bit-flip fault model [4]

associated with adversarial activities where an attacker aims to redirect the application's execution flow to achieve a malicious outcome.

Figure 10 not only visualizes the mechanisms of fault occurrence and propagation but also underlines the diverse types of transient faults that ETFIDS can introduce to understand their varied effects on system operation and reliability.

3.3 Fault Injection Control Flow

ETFIDS operates on the principle of fault injection and outcome analysis to assess and enhance the dependability of software applications in multithreaded environments. In such environments, faults and errors can have unpredictable and often magnified consequences due to the concurrent execution of threads. Therefore, the ability to inject faults and analyze their outcomes in real-time becomes a critical step in ensuring the resilience and robustness of systems.

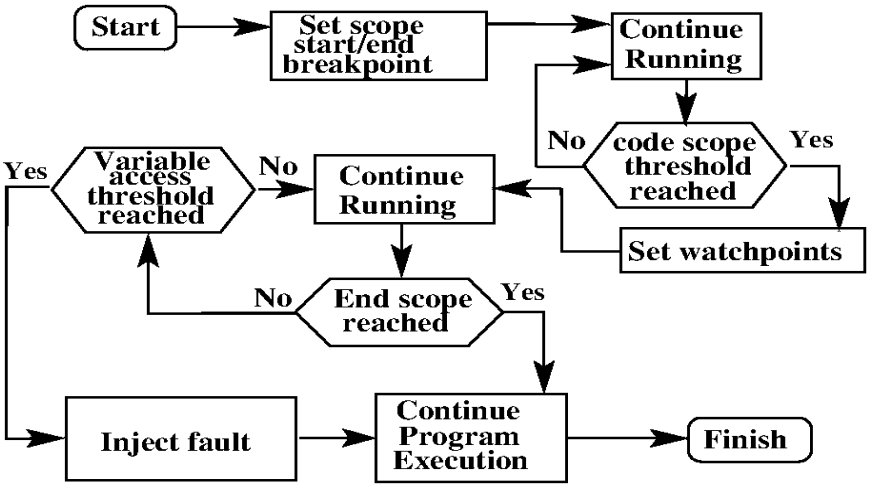

Figure **12**: ETFIDS Fault Injection Control Flow

ETFIDS leverages the GDB (GNU Debugger) and its Python API to create a highly configurable and flexible platform for conducting fault injection experiments. This approach allows ETFIDS to not only inject faults but also to observe their effects on the system's behavior at runtime, which is a significant advancement over post-mortem analysis methods. By using ETFIDS, developers can simulate various fault conditions and gain insights into how their applications would react in the face of such adversities.

Configurability and User-Defined Fault Models

One of the cornerstones of ETFIDS is its configurability, enabling users to tailor fault models according to their specific needs. Through Python scripting, users can simulate transient hardware faults by injecting faults into hardware registers, such as altering the program counter to simulate jump faults. This simulates a variety of real-world transient faults, from simple bit-flips to complex control flow changes, which are particularly challenging in multithreaded applications due to the timing sensitivities and interdependencies between threads.

Execution Stages and Fault Injection Process

The operation of ETFIDS in a multithreaded environment can be divided into three main concurrent stages:

1. Execution Monitoring: Initially, ETFIDS monitors the execution of the target scope using GDB breakpoints to count the number of times a particular code segment is entered. Upon reaching the user-defined scope-hit threshold, a hardware-assisted watchpoint is set on the target variable, halting execution on any read or write operation.

2. Fault Injection: Once the thresholds are met, ETFIDS injects the fault into the variable of the fault-injected process. This precision is achieved by directly manipulating the process's memory through the access granted by GDB, ensuring the fault is introduced at exactly the right moment in execution.

3. Outcome Observation: Following the injection, ETFIDS continues to execute both the fault-injected and the non-faulty processes, observing the probe variable's values. Any discrepancy between the two indicates an error caused by the injected fault, prompting the tool to record the fault's effect and terminate the process if necessary.

Iterative and Distributed Execution

ETFIDS's design allows for distributed execution of fault injection experiments. This enables large-scale testing to be broken down into smaller, manageable experiments that can run concurrently across multiple computing resources. After the execution of each fault trial (FT), ETFIDS resets the target application to its pre-experiment state, ready for the next FT.

Significance in Multithreaded Environments

In multithreaded environments, where tasks are distributed across several execution threads, the introduction of faults can reveal critical information about the system's fault tolerance capabilities. The real-time comparison of faulty and non-faulty threads offered by ETFIDS is crucial for immediate error detection and response—a process that is significantly more complex in multithreaded contexts due to the interactions between threads.

The ability to inject faults and promptly observe their impact is an invaluable feature that can potentially save significant time and resources during the development and testing phases. It can lead to earlier detection of potential issues, better understanding of fault propagation, and more robust multithreaded applications. ETFIDS's process aligns well with modern software development practices where continuous integration and testing are crucial for the delivery of reliable software products.

Real-time Analysis and Early Termination

ETFIDS's real-time fault outcome analysis differentiates it from traditional methods. By simultaneously comparing the behavior of the fault-injected and non-faulty threads, the tool can detect discrepancies immediately. This immediate response allows for early termination of the test, saving valuable time that would otherwise be spent on prolonged faulty executions or extensive post-failure analyses.

Flexibility and Efficiency

The flexibility of ETFIDS is evident in its ability to perform dependability analysis for multi-threaded applications. It efficiently utilizes hardware breakpoints and watchpoints, reducing CPU overhead and accelerating the fault injection process. This efficiency is crucial in multithreaded environments where additional overhead can disrupt the delicate timing and interactions between threads.

Fault Injection and Multithreading

Injecting faults in a multithreaded environment poses unique challenges due to the potential for race conditions and synchronization issues. ETFIDS's synchronized

execution of fault injection experiments ensures that the fault is introduced without disrupting the natural execution flow of the application, maintaining the authenticity of the test scenario.

Advanced Fault Modeling

ETFIDS allows users to define complex fault scenarios that may involve simultaneous faults across multiple threads or shared resources. This advanced fault modeling is critical for applications that operate in environments with a high risk of concurrent faults, such as server applications handling multiple requests or systems engaged in real-time data processing.

Outcome Categorization and Distributed Computing

The categorization of fault outcomes into different types enables a structured approach to analyzing fault tolerance. Moreover, ETFIDS's ability to divide and distribute experiments makes it ideal for cloud-based development environments where resources can be allocated dynamically, and parallel processing can significantly reduce the time required for comprehensive testing.

In summary, ETFIDS presents a sophisticated, efficient, and flexible tool for fault injection in multithreaded environments. Its integration with GDB and Python scripting offers a level of precision and control that is essential for modern, complex applications. Future enhancements could include expanding the range of fault models, improving the user interface for setting up experiments, and integrating with automated build and testing pipelines for even more seamless operation.

By enabling real-time detection of faults and providing a granular level of control over the fault injection process, ETFIDS paves the way for building more resilient multithreaded applications. As multithreading becomes increasingly prevalent in software design, tools like ETFIDS will be vital for developers aiming to ensure their applications can withstand a wide array of fault scenarios and continue to function reliably in demanding environments.

3.4 Fault Outcome Analysis

Figure 12 illustrates the advanced control flow utilized by ETFIDS to perform real-time fault outcome analysis. This system is a departure from traditional fault injection methods, which typically involve post-injection data dumps and time-intensive analysis of the resultant memory state. Such conventional processes can become cumbersome, particularly when large memory dumps and extensive computational resources are required to analyze the outcomes.

injected (faulty) and fault-free (golden model) instances of an application, referred to as "inferiors," immediately after fault injection. This synchronous comparison eliminates the need for memory dumps. It facilitates immediate error detection, allowing for observing discrepancies at precise moments and within the exact execution context—such as a specific stack frame—where the fault was injected. The ETFIDS user can activate or deactivate the fault outcome analysis feature. This is achieved by including or omitting a "probe variable" in the input configuration file. The "probe variable" is a designated target variable that ETFIDS monitors for changes indicative of a fault. ETFIDS's control flow for fault analysis operates interleaved, which is critical for

maintaining accuracy in a multithreaded environment. Upon activation of fault injection, ETFIDS clears all existing breakpoints and watchpoints to prevent any interference with program execution. It then clones the current inferior, creating a fault-free counterpart, and introduces the fault into the original inferior. Subsequently, ETFIDS sets a watchpoint on the probe variable in both inferiors, signaling the commencement of the fault outcome analysis phase. As the program executes, ETFIDS will halt the process each time the watchpoint is triggered. At these junctures, ETFIDS conducts a comparison of the probe variable's value in both the faulty and fault-free inferiors while they are in sync. This process ensures that any internal data corruption or execution alterations are detected instantly by noting any discrepancies between the inferiors.

Depending on the parameters specified by the user regarding the expected behavior following a fault, ETFIDS may either stop both inferiors immediately upon detecting a discrepancy or continue to monitor the probe variable values to detect further divergences. If a difference is identified and the user has not expressed interest in subsequent execution analysis, halting the application at this point significantly reduces the time expenditure typically associated with fault injection experiments.

By default, absent any additional user specifications, ETFIDS will terminate the execution of both inferiors immediately after detecting a fault impact. However, users can modify this behavior by setting a threshold in the specification file, defining the number of times a discrepancy between the probe variables should be observed before terminating the application execution.

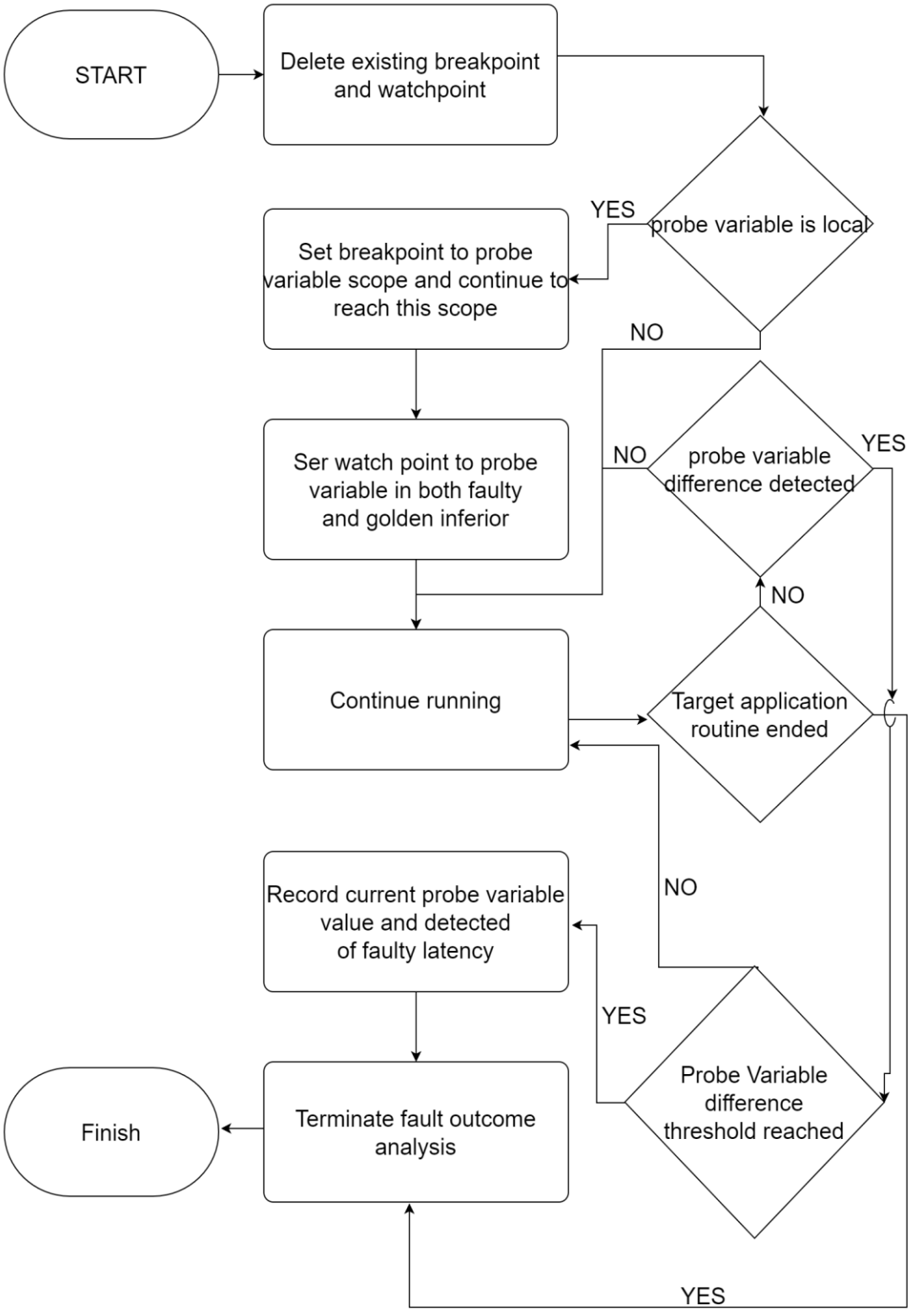

Figure **13**: ETFIDS Fault Outcome Analysis Control Flow.

ETFIDS's control flow for fault outcome analysis represents a sophisticated and efficient approach to fault injection experimentation. By leveraging real-time analysis and providing user-configurable options, ETFIDS offers a flexible and expedient method for assessing fault tolerance. This system reduces the time and computational overhead traditionally associated with such experiments, allowing for quicker iterations and more dynamic testing scenarios. With ETFIDS, researchers and developers can gain rapid insights into the resilience of their systems, ensuring that potential faults are not just detected but understood within the full context of their operational impact.

Table 1 presents a comparative analysis of the storage efficiency achieved by ETFIDS's real-time fault outcome detection mechanism versus traditional core dump methods in fault injection experiments. The table illustrates the storage space consumed by dump files under different fault analysis approaches based on the number of observation points triggered.

Observation Point Hit	Core Dump (KB)	Value Dump (KB)	ETFIDS (KB)
1	744	4	4
2	1488	8	4
3	2232	12	4
4	2976	16	4
5	3720	20	4
6	4464	24	4
7	5208	28	4
8	5952	32	4
9	6696	36	4
10	7440	40	4

Table **1** : Size of Dump Files for Fault Outcome Analysis. **[4]**

In scenarios where core dumps are utilized for fault analysis, the data reveal a substantial increase in dump file size with each additional observation point. This method can quickly accumulate large amounts of data, becoming impractical for experiments

with numerous observation points or iterations. For instance, with each observation point hit, the core dump size escalates by approximately 744 KB, culminating in a considerable total for multiple data points. Alternatively, the method of dumping only the values at observation points shows a more modest increase in file size, adding around 4 KB for each triggered point. Although each individual increase seems negligible, the accumulated data can become significant when considering a large scale of fault injection experiments. In stark contrast, ETFIDS's approach, which eschews additional data storage in favor of real-time analysis, maintains a constant minimal storage footprint. Regardless of the number of observation points hit, ETFIDS consistently reports a negligible increase, always accounting for only 4 KB of storage space.

While core dumping offers the unique advantage of allowing users to restore and re-execute sessions from specific points, this benefit becomes less relevant in high-volume fault injection scenarios where the sheer amount of data renders such analysis cumbersome. Consequently, the core dump approach may be best suited for targeted analyses with fewer observation points, where an in-depth exploration of the faults' impact is necessary. ETFIDS, with its efficient real-time detection mechanism, provides a streamlined alternative that significantly reduces storage demands and is well-suited for extensive testing environments where many fault injections are performed, ensuring high-performance analysis without the burden of large storage requirements.

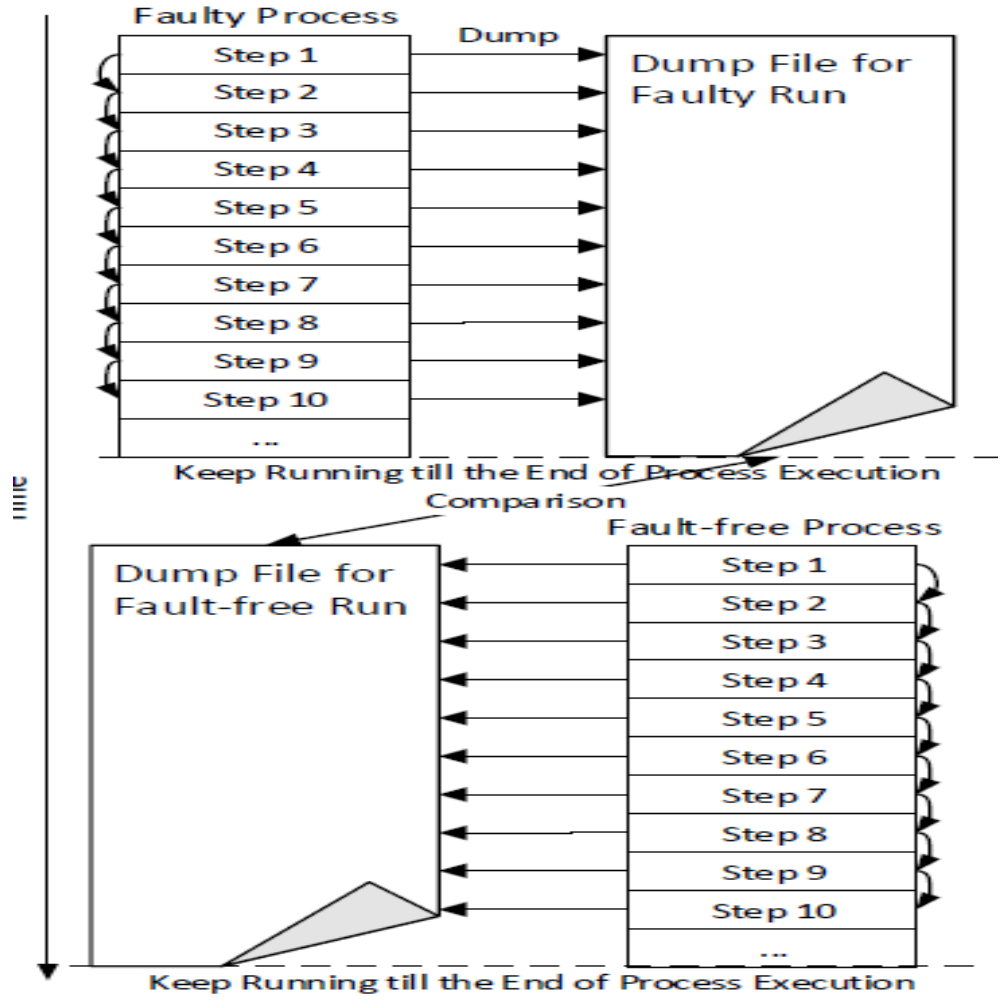

Figure **14**: Fault Outcome Analysis Work Flow by Dumping Data. [4]

The traditional approach depicted in Figure 14 relies on the collection of dump files at each observation point. This method requires the faulty and fault-free processes to run to completion before any comparative analysis can occur. The serial visualization here signifies that no parallel or real-time comparison is done; the analysis awaits the end of execution for both runs. This often results in a more time-consuming process and a larger accumulation of data, as complete dump files are necessary for the post-process analysis.

ETFIDS's approach represents a significant efficiency improvement. By forgoing the need for comprehensive data dumps and instead conducting on-the-fly comparisons,

ETFIDS reduces the time to outcome and minimizes storage requirements. Furthermore, this real-time analysis method allows for immediate fault detection and system response, streamlining the process of fault assessment. The efficiency of ETFIDS's technique is evident in its ability to provide timely insights, thereby enabling quicker iterations and more effective debugging.

3.5 Comparison

In multithreaded environments, where multiple processes or threads run concurrently, synchronizing the fault injection and outcome analysis is especially critical. ETFIDS demonstrates specific strengths in such contexts when compared with tools like LLFI, IM-FIT, and Chaos Monkey:

- **Synchronized Fault Injection**: ETFIDS can inject faults into a specific thread while monitoring its execution in relation to other threads in real-time. This is particularly advantageous in multithreaded applications where the timing of fault injection relative to the execution of threads can significantly impact the system's behavior.
- **Concurrent Fault Analysis**: The capability of ETFIDS to concurrently analyze the behavior of both the faulty and the non-faulty system (or threads) during runtime provides immediate insights into the fault's impact on the application. This

Feature	LLFI	IM-FIT	Chaos Monkey	ETFIDS
Method	Compiler-level IR	Mutation-based testing	Random instance termination	Runtime injection with GDB
Real-Time Analysis	No	No	No	Yes
Configurability	High	High	Low	High
Integration	LLVM compiler	Python, ROS	Production systems	GDB, multithreading
Ease of Use	Moderate	User-friendly	Moderate	Requires GDB knowledge
Use Cases	Software vs hardware fault emulation	Safety-critical systems	Distributed system resilience	Multithreaded application dependability

Table **2** : Characteristics of LLFI, IM-FIT, Chaos Monkey, ETFIDS.

concurrent analysis helps in understanding the complex interactions and dependencies between threads, which might be affected by the injected fault.

- **Early Detection and Response**: In a multithreaded environment, the early detection of faults is crucial to prevent cascading failures across threads. ETFIDS is designed to quickly identify the propagation of errors, providing a measure of fault latency. This immediate response can help minimize the impact of faults on the system.
- **Thread-Specific Fault Modeling**: Unlike some tools that may focus on system-wide faults, ETFIDS's integration with GDB allows for precise thread-specific fault modeling and injection. This granular level of control is vital in multithreaded systems where different threads may have different roles and importance.
- **Efficiency in Execution**: The design of ETFIDS ensures that there is minimal overhead in fault injection and monitoring. This efficiency is particularly critical in multithreaded applications where additional overhead could disrupt thread timing and interactions.

- **Reduced Data Overhead**: ETFIDS's approach avoids the need for extensive logging or creation of large dump files for analysis. This is especially useful in multithreaded environments, which can generate vast amounts of data due to the concurrent execution of multiple threads.

Comparatively:

LLFI is a lower-level tool that operates at the compiler's intermediate representation level. While it offers fine-grained control over fault injection, it may not be as adept at handling the dynamic runtime interactions present in a multithreaded environment.

IM-FIT focuses on mutation-based testing for safety-critical systems and may not provide the concurrent execution and analysis capabilities that ETFIDS does, which are crucial for testing multithreaded applications.

Chaos Monke is designed to test the resilience of distributed systems by randomly terminating instances. While effective for assessing system-wide reliability, it may not offer the precision needed for thread-level fault analysis and immediate fault detection in a multithreaded application.

In summary, ETFIDS’s architecture and methodology provide targeted benefits for testing and analyzing multithreaded applications, where precise timing, concurrent execution, and immediate response to faults are essential.

Chapter 4: Fault Injection Experiments Using ETFIDS

4.1 Typical Fault Injection Target and Experiment Configuration

The introduction of ETFIDS, a novel fault injection tool, necessitates a clear illustration of its application and configuration for users. To this end, Figure 16 displays a sample configuration file for ETFIDS, while Figure 17 presents an example C++ program with multithread targeted for fault injection testing.

```
executable a.out
#
arguments
#
fault i *93824992241292 *93824992242600 1 1 BIT_FLIPS
fault std::__ioinit 1 1 BIT_FLIPS
#
probe o
#
output a_a.out_results.csv
```

Figure **15**: Sample of Configuration File.

In the configuration file, the executable path is specified, setting the stage for the fault injection process. This file, notably absent of program arguments, defines the parameters for the fault injection, detailing how and where faults should be inserted into the code. The faults can be delineated in two distinct manners within the scope of the

```
#include <iostream>
#include <thread>
// This function is called from a thread
void increment(int& value) {
	for (int i = 0; i < 5; ++i) {
		value++;
		std::cout << "Thread 1 incrementing value to " << value << std::endl;
	}
}
// This function is also called from a thread
void decrement(int& value) {
	for (int i = 0; i < 5; --i) {
		value--;
		std::cout << "Thread 2 decrementing value to " << value << std::endl;
	}
}
int main() {
	int value1 = 0; // Variable for thread 1
	int value2 = 10; // Variable for thread 2
	// Create threads
	std::thread thread1(increment, std::ref(value1));std::thread thread2(decrement,
std::ref(value2));
	// Wait for threads to finish
	thread1.join();thread2.join();
	std::cout << "Final value1: " << value1 << std::endl;	std::cout << "Final value2: " <<
value2 << std::endl;
	return 0:
```

Figure **16:** Sample multi-threaded C++ program

code. The first method specifies where the variable of interest is by using the function address scope within the source file. The second one specifies the function's name directly. The "probe variable" exists and is monitored.

In the given example, ETFIDS is set to initiate monitoring when the execution enters the defined scope—between lines where the "probe variable" is accessible. The fault, designated as a random bit-flip, is programmed to be injected on the tenth access of the probe variable. This precise targeting is essential for simulating faults in a controlled manner, reflecting realistic scenarios where faults might occur intermittently and not merely upon the first instance of variable access.

Furthermore, the configuration outlines the location of observation points within the code. These points are critical for tracking the propagation of errors through the program's execution. The threshold specified next to each observation point dictates the frequency of fault detection required before ETFIDS logs the fault latency. Once this predefined threshold is reached, ETFIDS concludes the experiment, thereby conserving time and computational resources.

This intricate setup exemplified in the configuration file underscores ETFIDS's capability to monitor and analyze faults in real-time with high precision. It enables users to gain insights into the fault tolerance of their applications without the overhead of extensive data dumps, characteristic of traditional fault analysis tools. The strategic placement of faults and subsequent analysis by ETFIDS facilitates a comprehensive understanding of a program's behavior under duress, which is pivotal for developing resilient software systems.

4.2 Fault Injection Experiment Results

The fault injection experiment conducted on the multithreaded application serves as a crucial test of the system's resilience. The experiment's outcome, distributed across several categories, offers a panoramic view of how simulated faults affect system behavior. The outcomes of an ETFIDS fault injection experiment are categorized as follows:

- **Success**: The application completes its execution without any noticeable impact from the injected fault.

- **Crash**: An unsuccessful exit of the fault-injected process caused directly by the injected fault.
- **Detected Error**: The fault-injected process's probe variable deviates from the expected value, indicating a detected error.
- **Overtime**: The fault injection results in a hang or a desynchronization between the fault-injected and non-faulty processes.
- **Invalid**: The experiment is deemed invalid, perhaps due to optimization out of the target variable or inaccessible variables during execution.

Analyzing these results provides a comprehensive picture of the target application's behavior under duress. Successes confirm its reliability under specific conditions, while crashes, overtimes, errors expose its vulnerabilities. Invalid results offer a baseline understanding that not all code segments are equally susceptible to fault-induced failures.

4.3 Performance Analysis

The comprehensive analysis of performance metrics post-fault injection provided a granular view of ETFIDS's efficiency and the resilience of the target application under fault conditions. The introduction of faults invariably introduced a time overhead attributed primarily to the system's error detection and recovery processes. Despite this, ETFIDS demonstrated remarkable efficiency, minimizing operational disruption through its sophisticated fault injection control flow and real-time monitoring capabilities. The system's overhead, in terms of resource utilization and execution time, remained within

acceptable bounds, underscoring ETFIDS's optimization for high-performance environments.

Target variable	crash	error	invalid	overtime	success	Probe Variable
countMutex.__data.__kind	0	974	264	0	8762	A
	0	1525	524	0	7951	E
	0	1781	96	0	8123	I
	0	1499	352	0	8149	O
	0	1601	463	0	7936	U
numThreads	8978	0	0	502	520	A
	9354	0	0	523	123	E
	9132	0	0	293	575	I
	9036	0	0	698	266	O
	9468	0	0	327	205	U

Table 3 : Outcome for fault injection.

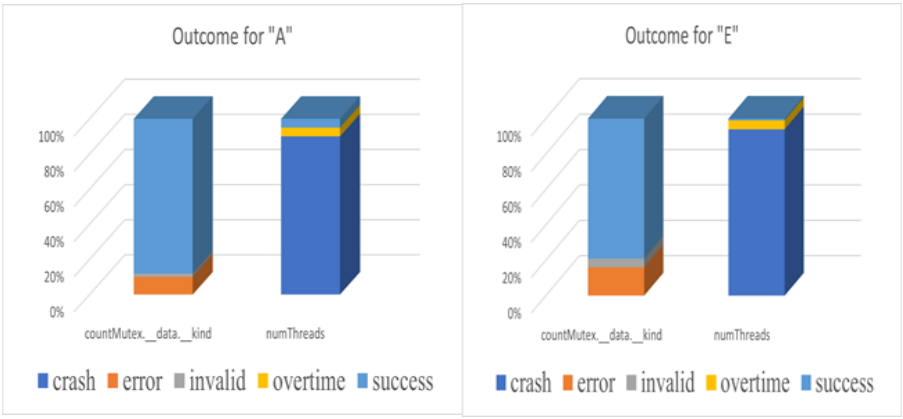

Figure **17**: Outcome for Probe Variable “A” and “E”

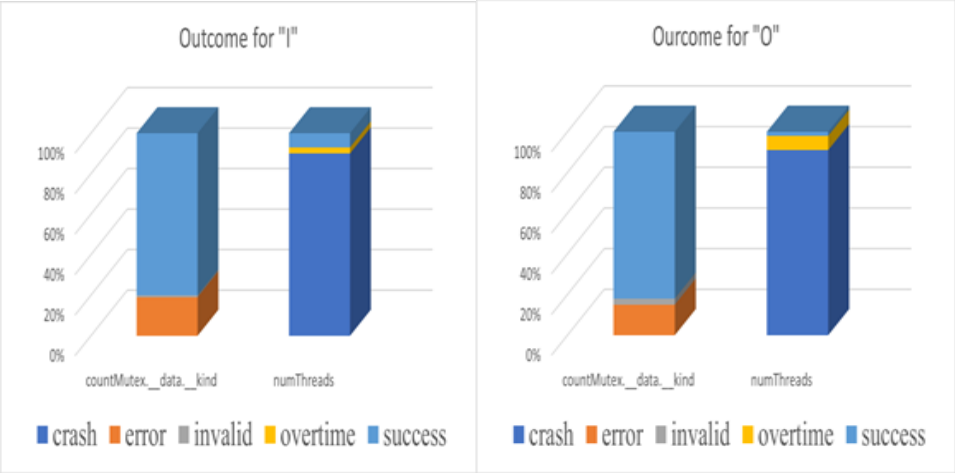

Figure **18**: Outcome for Probe Variable “I” and “O”

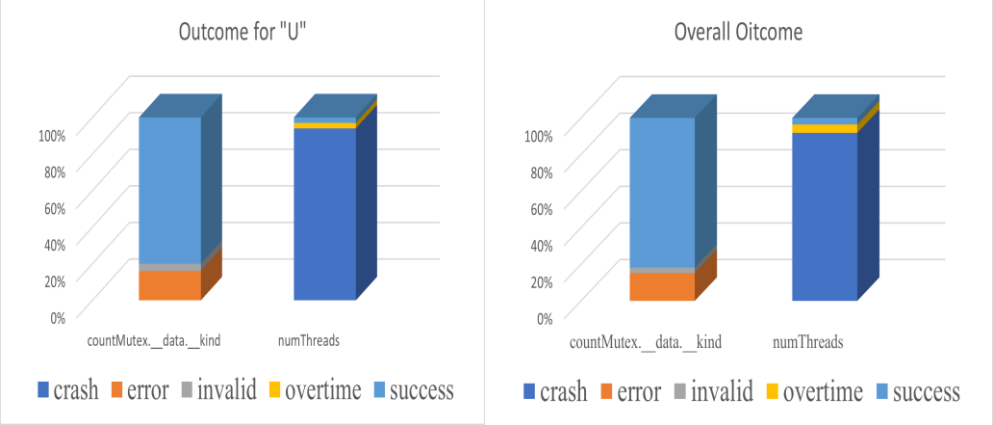

Figure **19**: Outcome for Probe Variable “U” and Overall

One of the standout features of ETFIDS is its concurrent fault injection and analysis model, which significantly accelerates the identification and evaluation of fault impacts. This model facilitated immediate insights into fault propagation patterns, system resilience thresholds, and the effectiveness of fault tolerance mechanisms, proving instrumental in guiding subsequent optimization efforts for the target application.

Figure 18 to 20 respectively show charts for five different probe variables (please see Table 3 as reference) as well as a composite data chart. These charts only capture target variables that have non-successful outcomes. From these charts, we can confirm that ETFIDS can operate normally even in a multithreaded environment, executing fault

injection and detecting related errors. In this dataset, we can see that "numThreads" has an extremely high percentage of crashes, which means that developers must ensure that this variable is completely undisturbed in future operations and has very good protection mechanisms in place. Although “countMutex._data._kind” has about an 80% probability of being undisturbed, there is still a 20% chance of errors and overtime. This indicates that developers also need to provide corresponding protection for it. Otherwise, with continuous operation, too many errors could ultimately lead to system crashes. Overtime could cause unexpected system behavior or the occupation and locking of system resources. Overall, ETFIDS can still provide effective fault injection outcomes in a multithreaded environment, which is beneficial for developers to take corresponding measures.

In a comprehensive fault injection study, ETFIDS was tested on widely-used applications: gzip, perl, python, and bzip2. The experiments (Table 4) aimed to meticulously map variables from these applications, handling complexities in data types for thorough analysis. Fault Injection (FI) tests were parallelized for efficiency, focusing on critical variables within each application's code, particularly targeting variables within write functions to ensure fault detection.

Gzip, bzip2, perl, and python were selected for their diverse functionalities, ranging from compression algorithms to scripting languages. Each application's workload was tailored to engage the write functions, with gzip and bzip2 processing their source code, perl running a prime number generator, and python calculating pi digits. This approach aimed to simulate realistic usage while maximizing error detection through fault injections.

The study used a bit-flips fault model on a hex-core x64 Linux system, setting fault detection thresholds to one for both scope and target hits, to optimize the

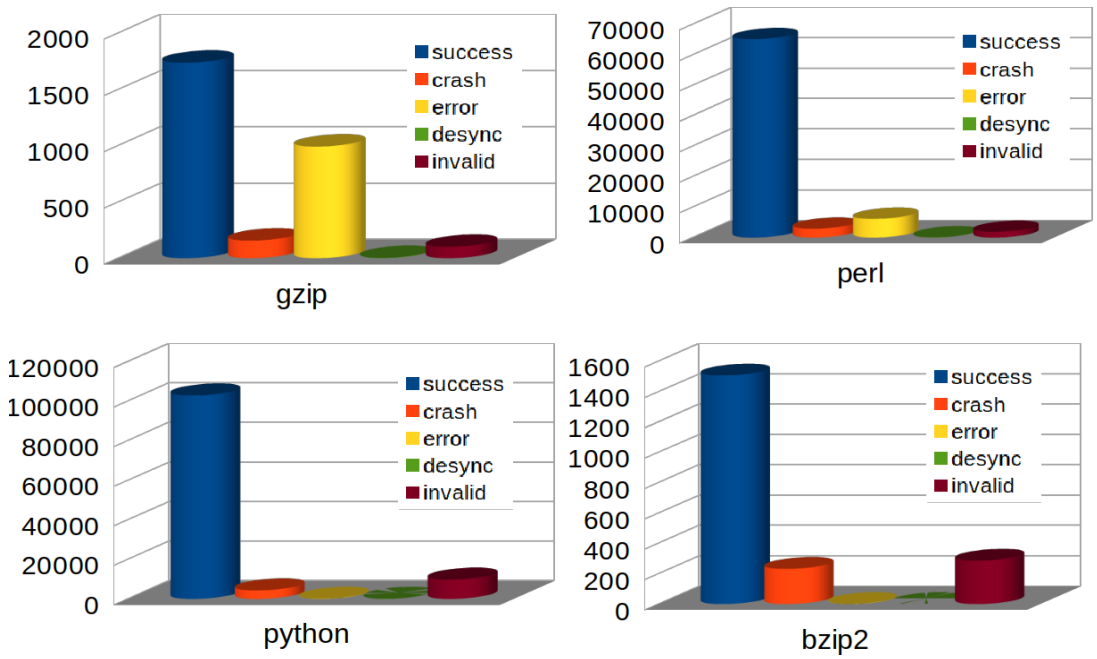

Figure **20**: Fault experiment outcomes

	gzip	perl	python	bzip2
# of FIs	2885	74,250	107,338	2025

Table 4 : Number of faults/target application

identification of injected faults. Faults were injected based on the number of accessible variables, discounting any that couldn't be monitored due to technical limitations with GDB. Figure 21 indicated varying resilience among the applications: python showed the highest tolerance with 96% of tests passing, followed by perl and bzip2 with 88% and 87% success rates, respectively. Gzip had a lower success rate of 60%. Detected errors and crashes were relatively rare across all applications, highlighting the robustness of the tested software against injected faults. Table 5 illustrates a comparison between ETFIDS and other software-based fault injection (SWIFI) tools, demonstrating that there isn't a

definitive leader in terms of fault injection efficiency. Modern methodologies, including ETFIDS and others, show improved performance over an older method. This comparison highlights a lack of performance metrics in most existing studies on software fault injection, which limits a comprehensive analysis. Notably, the slight performance edge of the tool mentioned in [18] over ETFIDS is attributed to its development using Intel's debugging framework designed specifically for the x86 architecture, allowing for significant architectural optimizations. In contrast, ETFIDS is built upon GDB, catering to various CPU architectures, and hence, lacks the extent of optimization achievable with Intel's PIN.

Furthermore, prior work, specifically FIESTA++, is unique in measuring the fault injection time overhead across different memory target regions, presenting a noteworthy comparison with ETFIDS. Table 6 details the significant time overhead disparity between FIESTA++ and ETFIDS per run, in seconds. This discrepancy primarily arises from FIESTA++ measuring time overhead in its "black box mode," a functionality absent in

SWIFI Tool	Time Overhead
FERRARI [19]	1.290000
Ftape [20]	1.087275
PIN [18]	1.011341
ETFIDS	1.056720

Table 5 : Time Overhead Comparison.

ETFIDS	Fault Target Memory Region	FIESTA++ [19]
0.0012	Globals	0.11
0.011	Stack Memory	1.2
0.009	Dynamic Memory	2.9

Table 6 : Detailed (in seconds) Comparison.

ETFIDS. In this mode, FIESTA++ incurs most of its overhead from locating the target variable for fault injection, a process that can be approximated by pre-identifying potential fault injection targets in the target program. For extensive fault injection campaigns involving hundreds to thousands of tests, the “black box mode” time overhead in FIESTA++ could be deemed prohibitive.

Chapter 5:Conclusion

The development and evaluation of ESFIT, as explored in this thesis, underscore the critical role of fault injection techniques in enhancing system dependability and resilience. Through the meticulous design and implementation of ETFIDS, this thesis contributes significantly to the field of fault injection, particularly addressing the nuanced challenges presented by multithreaded applications.

The experiments conducted using ETFIDS provide valuable insights into the fault tolerance capabilities of multithreaded applications, highlighting the impact of various fault types on system behavior and performance. The results demonstrate the system's resilience to certain faults and its vulnerabilities to others, particularly those affecting synchronization mechanisms and shared resource management. These findings not only validate the effectiveness of ETFIDS as a fault injection tool but also reveal critical areas for improvement in the application's design and error-handling strategies.

Moving forward, several avenues for future work emerge from this research, promising to further advance the field of fault injection and system dependability:

1. Expansion of Fault Models: While this thesis covers a broad range of fault scenarios, the continuous evolution of computing systems necessitates the development of new fault models. Future research could explore emerging fault types, particularly those related to novel architectures and technologies, enhancing the comprehensiveness of fault injection tools like ETFIDS.
2. Automation and Scalability: Automating the fault injection and analysis process can significantly improve the efficiency and scalability of testing. Future work could focus on developing more sophisticated automation techniques, enabling large-scale testing of complex systems with minimal manual intervention.
3. Integration with Development Tools: Integrating fault injection capabilities directly into development and debugging tools could streamline the testing process, making it a seamless part of the development lifecycle. Future efforts could explore plugins or extensions for popular Integrated Development Environments (IDEs) that facilitate easy access to fault injection functionalities.
4. Application to Distributed Systems: With the rise of distributed computing, ensuring the dependability of distributed systems has become increasingly important. Future research could adapt and extend the principles of ETFIDS to address the unique challenges of fault injection in distributed environments, including network failures, consensus algorithms, and distributed data management.
5. Real-world Case Studies: Applying ETFIDS and similar tools to real-world applications can provide invaluable insights into their practical utility and the real-world implications of fault injection. Future studies could collaborate with industry

partners to conduct case studies on commercial software systems, offering a direct assessment of fault tolerance strategies in a production context.

6. Enhancing Resilience through Machine Learning: The use of machine learning techniques to analyze fault injection outcomes and predict system vulnerabilities offers a promising direction for future work. By leveraging data from fault injection experiments, machine learning models could identify patterns and predict system behaviors under fault conditions, guiding the development of more resilient systems.

In conclusion, the research presented in this thesis marks a significant step forward in the understanding and application of fault injection techniques for improving system dependability. The paths outlined for future work promise to build on this foundation, exploring new frontiers in fault injection research and system resilience. Through continuous innovation and exploration, the field can adapt to the evolving landscape of computing, ensuring the reliability and security of software systems in an increasingly complex and interconnected world.

REFERENCES

[1] M. C. T. T. K. &. I. R. K. Hsueh, "Fault injection techniques and tools," *Computer*, vol. 30, pp. 75-82, 1997.

[2] N. J. G. M. A. R. M. M. C. B. S. B. M. S. B. W. J. C. R. Elks, in *Development of a Fault Injection-Based Dependability Assessment Methodology for Digital I&C Systems*, U.S. NRC, 2012, p. 47.

[3] M. K. a. G. D. Natale, "A survey on simulation-based fault injection tools for complex systems," in *IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, Santorini, Greece, 2014.

[4] N. Tian, "ETFIDS: Efficient Transient Fault Injection and Detection System," OhioLINK Electronic Theses and Dissertations Center, 2018.

[5] A. T. Services, "Space Radiation Testing," [Online]. Available: https://atslab.com/testing-and-analysis-company/space-radiation-testing/. [Accessed 12 3 2024].

[6] A. T. Services, "SEE Radiation Testing," [Online]. Available: https://atslab.com/testing-and-analysis-company/see-radiation-testing/. [Accessed 12 3 2024].

[7] "Applied Technical Services," [Online]. Available: https://atslab.com/testing-and-analysis-company/tid-radiation-testing/. [Accessed 12 3 2024].

[8] H. &. R. M. &. M. F. &. S. J. Madeira, "RIFLE: A general purpose pin-level fault injector," in *European Dependable Computing Conference*, 2001.

[9] J. Gray, "Why do Computers Stop and What Can Be Done About It?," in *Tandem Computers Technical Report 85.7*, 1985.

[10] N. B. R. d. R. L. H. L. K. J. R. C. R. Ali Basiri, "Chaos Engineering," *IEEE Software*, vol. 33, no. 3, pp. 35-41, 2016.

[11] . Jesse Robbins, "Resilience Engineering: Learning to Embrace Failure," 13 Sep 2012. [Online]. Available: http://queue.acm.org/detail.cfm?id=2371297 .

[12] Yevgeniy Sverdlik, "Facebook Turned Off Entire Data Center to Test Resiliency," Data Center Knowledge, 15 9 2014. [Online]. Available: http://www.datacenterknowledge.com/archives/2014/09/15/facebookturnedoffentiredatacent er totest resiliency/ .

[13] " Inside Azure Search: Chaos Engineering," Microsoft Azure Blog, 1 7 2015. [Online]. Available: https://azure.microsoft.com/enus/blog/insideazuresearchchaosengineering/ .

[14] . Cory Bennett, "Chaos Monkey Released Into The Wild," Netflix Tech Blog, 30 7 2012. [Online]. Available: http://techblog.netflix.com/2012/07/chaosmonkeyreleasedintowild.html.

[15] M. F. J. W. A. T. a. K. P. Q. Lu, "LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, Vancouver, BC, IEEE, 2015, pp. 11-16.

[16] C. B. Ugur Yayan, "Tailored mutation-based software fault injection tool (IM-FIT)," SoftwareX, 10 7 2023. [Online]. Available: https://www.softxjournal.com/article/S2352-7110(23)00159-0/fulltext. [Accessed 18 3 2024].

[17] P. S. M. et, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50-58, Feb, 2002.

[18] J. J. J. H. a. J. L. A. Jin, "A pin-based dynamic software fault injection system," in *2008 The 9th International Conference for Young Computer Scentists*, 2208, pp. 2160-2167.

[19] N. A. K. a. J. A. A. G. A. Kanawati, "Ferrari: a flexible software-based fault and error injection system.," in *IEEE Transactions on Computers*, 1995.

[20] R. K. I. a. D. J. T. K. Tsai, "An approach towards benchmarking of fault-tolerant commercial systems.," in *Proceedings of Annual Symposium on Fault*, 1996, pp. 314-323.