# FIDL: A Fault Injection Description Language for Compiler-based SFI Tools

Maryam Raiyat Aliabadi and Karthik Pattabiraman

Electrical and Computer Engineering, University of British Columbia
Vancouver, BC, Canada
{raiyat, karthikp}@ece.ubc.ca

**Abstract.** Software Fault Injection (SFI) techniques play a pivotal role in evaluating the dependability properties of a software system. Evaluating the dependability of software system against multiple fault scenarios is challenging, due to the combinatorial explosion and the advent of new fault models. These necessitate SFI tools that are programmable and easily extensible. This paper proposes FIDL, which stands for fault injection description language, which allows compiler-based fault injection tools to be extended with new fault models. FIDL is an Aspect-Oriented Programming language that dynamically weaves the fault models into the code of the fault injector. We implement FIDL using the LLFI fault injection framework and measure its overheads. We find that FIDL significantly reduces the complexity of fault models by 10x on average, while incurring 4-18% implementation overhead, which in turn increases the execution time of the injector by at most 7% across five programs.

## 1 Introduction

Evaluating the dependability properties of a software system is a major concern in practice. Software Fault Injection (SFI) techniques assess the effectiveness and coverage of fault-tolerance mechanisms, and help in investigating the corner cases [4, 5, 15]. Testers and dependability practitioners need to evaluate the software system's dependability against a wide variety of fault scenarios. Therefore, it is important to make it easy to develop and deploy new fault scenarios [18].

In this paper, we propose FIDL (Fault Injection Description Language)[1], a new language for defining fault scenarios for SFI. The choice of introducing a specialized language for software fault injection is motivated by three reasons. First, evaluating the dependability of software system against multiple fault scenarios is challenging - the challenge is combinatorial explosion of multiple failure modes [11] when dealing with different attributes of a fault model (e.g, fault types, fault locations and time slots). Second, due to the increasing complexity of software systems, the advent of new types of failure modes (due to residual software bugs) is inevitable [5]. Previous studies have shown that anticipating and modeling all types of failure modes a system may face is challenging [11].

[1] Pronounced *Fiddle* as it involves fiddling with the program.

Hence, SFI tools need to have extensibility facilities that enable dependability practitioners to dynamically model new failure modes, with low effort. Third, decoupling the languages used for describing fault scenarios from the fault injection process enables SFI tool developers and application testers to assume distinct roles in their respective domains of expertise.

The main idea in FIDL is to use Aspect Oriented Programming (AOP) to weave the aspects of different fault models dynamically into the source program through compiler-based SFI tools. This is challenging because the language needs to capture the high-level abstractions for describing fault scenarios, while at the same time being capable of extending the SFI tool to inject the scenarios. Prior work has presented domain specific languages to drive the fault injection tool [3, 6, 11, 16, 18]. However, these languages provide neither high level abstractions for managing fault scenarios, nor dynamic extensibility of the associated SFI tools. *To the best of our knowledge, FIDL is the first language to provide high-level abstractions for writing fault injectors spanning a wide variety of software faults, for extending compiler-based SFI tools.*

**Paper contributions:** The main contributions of this paper are as follows:

- Proposed a fault injection description language (FIDL) which enables programmable compiler-based SFI tools.
- Built FIDLFI, a programmable software fault injection framework by adding FIDL to LLFI, an open-source, compiler-based framework for fault injections [1, 14].
- Evaluated FIDL and FIDLFI on five programs. We find that FIDL reduces the complexity of fault models by 10x on average, while incurring 4 to 18% implementation overhead, which in turn increases the time overhead by at most 6.7% across programs compared to a native C++ implementation.

## 2 Background

We developed FIDL as an Aspect-Oriented Programming (AOP) language on the LLFI fault injection framework. In this section, we first provide a brief overview of LLFI. We then explain why we are motivated to develop an AOP language for extending and driving LLFI. Though we demonstrate FIDL in the context of LLFI, it can be applied to any compiler-based SFI tool.

### 2.1 LLFI

LLVM is a production, open-source compiler that allows a wide variety of static program analysis and transformations [13]. LLFI is an open source LLVM-based fault injection tool that injects faults into the LLVM Intermediate Representation (IR) level of application source code [21]. LLFI was originally developed for hardware fault injection. It injects a fault (e.g., bit flip) into a live register at every run of program in specific locations that are instrumented during compile time [14]. LLFI also allows user to track the fault propagation path, and map it back to the application source code.

Since its development, we have extended LLFI to inject different kinds of software faults in a program in addition to hardware faults [1]. This is the version of LLFI that we use in this paper for comparison with FIDL.

### 2.2 Aspect-Oriented Programming (AOP)

Object-Oriented Programming (OOP) is a well-known programming technique to decompose a system into sets of objects. However, it provides a static model of a system - thus any changes in the requirements of software system may have a big impact on development time. Aspect-Oriented Programming (AOP) presents a solution to the OOP challenge since it enables the developer to adopt the code that is needed to add secondary requirements such as logging, exception handling without needing to change the original static model [17]. In the following, we introduce the standard terminology defined in AOP [17].

- **Cross-cutting concerns:** are the secondary requirements of a system that cut across multiple abstracted entities of an OOP. AOP aims to encapsulate the cross-cutting concerns of a system into aspects and provide a modular system.
- **Advice:** is the additional code that is "joined" to specific points of program or at specific time.
- **Point-cut:** specifies the points in the program at which advice needs to be applied.
- **Aspect:** the combination of the point-cut and the advice is called an aspect. AOP allows multiple aspects to be described and unified into the system automatically.

## 3 Related work

A wide variety of programmable fault injection tools based on SWIFI (SoftWare Implemented Fault Injection) techniques have been presented in prior work [3, 6, 11, 11, 12, 16, 18, 23]. In this section, we aim to define where FIDL stands in relation to them. More particularly, we argue why *"Programmability"* is a necessity for fault injection tools.

Programmability, is defined as the ability of programming the fault injection mechanism for different test scenarios based on desired metrics of the tester [6, 18]. Programmability has two aspects. The first is a unified description language that is independent of the language of the SFI tool [3]. This language is needed to accelerate the process of fault scenario development, and dynamically manage the injection space for a variety of fault types. The second aspect of programmability is providing high level abstractions in the language. The abstracted information keeps the fault description language as simple as possible. By removing the complexity of fault scenario's developing phases, high level abstraction enhances the usability of the tool [3, 9, 11, 16].

There have been a number of languages for fault injection. FAIL* is a fault injection framework supported by a domain specific language that drives the fault load distributions for Grid middleware [18]. FIG is supported by a domain specific language that manages the errors injected to application/shared library boundary [3]. Orchestra and Genesis2 use scripts to describe how to inject failures into TCL layers and service level respectively [6, 12]. LFI is supported by a XML-based language for introducing faults into the libraries [16]. EDFI is a LLVM-based tool supporting a hybrid (dynamic and static) fault model description in a user-controlled way through command line inputs [8]. However, the aforementioned languages do not provide high level abstractions, and hence developing a new fault model (or scenario) is non-trivial. PREFAIL proposes a programmable tool to write policies (a set of multiple-failure combinations) for testing cloud applications [11]. Although its supporting language provides high level abstractions, the abstracted modules only manage the failure locations, and do not provide any means to describe new failure types.

## 4 System overview

In this paper, we present FIDLFI: a programmable fault injection framework, which improves upon the previous work in both extensibility and high level abstraction. FIDLFI enables programmability of compiler-based SFI tools, and consists of two components: a SFI engine to manage fault injection, and FIDL as SFI driver to manage fault scenarios. It enables testers to generate aggregate fault models in a systematic way, and examine the behavior of the Application Under Test (AUT) after introducing the fault models.

We built the FIDL language to be independent from the language used in the fault injector, which is C++. This enables decoupling the SFI engine and FIDL. Figure 1 indicates the FIDLFI architecture, and the way both pieces interact with each other. The tester describes a fault scenario (new failure mode or a set of multiple failure modes' combinations) in FIDL script, and feeds it into the FIDL core, where it is compiled into a fault model in the C/C++ language. The generated code is automatically integrated into the SFI engine's source code. It enables the SFI engine to test AUT using the generated fault model.

In the rest of this section, we first explain how we design aspects in FIDL to specify the fault model, and then, present the algorithm to weave the models into the fault injector.

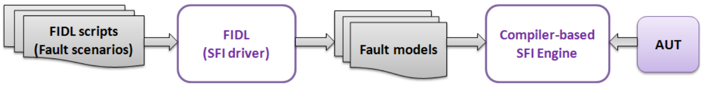

Fig. 1: FIDLFI architecture

### 4.1 FIDL Structure

FIDL represents the fault models in a granular fashion. We reflected the granularity in the fault model by designing it in the form of distinguishable modules, in which the associated attributes are described. The main attributes of a fault model includes fault type (what to inject), and fault location (where/when to inject) that forms the basis of the model.

A FIDL script is formed of four core entities; Trigger, Trigger*, Target and Action, each of which represents a specific task toward fault model design. Once a FIDL script is executed, the FIDL algorithm creates two separate modules (fault trigger and fault injector). Trigger, Trigger* and Target are entities which are representative for responding to the *where to inject* question in fault model design. For simplicity, we call all three entities as *Triggers*. Triggers provide the required information for FIDL algorithm to generate fault trigger module. Triggers are like programmable monitors scattered all over the application in desired places to which FIDL can bind a request to perform a set of Actions. An Action entity represents what to be injected in targeted locations, and is translated to fault injector module by the FIDL algorithm.

We use the terms *instruction* and *register* to describe the entities, as this is what LLVM uses for its intermediate representation (IR) [13]. The FIDL language can be adapted for other compiler infrastructures which use different terms.

**Trigger** identifies the IR instructions of interest which have previously been defined based on the tester's primary metrics or static analysis results.

*Trigger: <instruction name >*

**Trigger*** selects a special subset of identified instructions based on the tester's secondary metrics. This entity enables the tester to filter the injection space to more specific locations. Trigger* is an optional feature that is used when the tester needs to narrow down the Trigger-defined instructions based on specific test purposes, e.g, if she aims to trigger the instructions which are located in tainted paths.

*Trigger*: <specific instruction indexes>*

**Target** identifies the desired register(s) in IR level (variable or function argument in the source code level).

*Target: < function name :: register type >*

Register type can be specified as one of the following options;

*dst/RetVal/src (arg number)*

in which *dst* and *src* stand for destination and source registers of selected instruction respectively, and *RetVal* refers to the return value of the corresponding instruction. For example *fread:: src 2* means *entry into 3rd source register* *of fread instruction*, and similarly *src 0* means *entry into 1st source register of every Trigger-defined instruction.*

**Action** defines what kind of mutation is to be done according to the expected faulty behavior or test objectives.

*Action: Corrupt/Freeze/Delay/SetValue/ Perturb*

*Corrupt* is defined as bit flipping the Data/Address variables.*Delay* and *Freeze* are defined as creating an artificial delay and creating an artificial loop respectively, and *Perturb* describes an erroneous behavior. If Action is specified as Perturb, it has to be followed by the name of a built-in injector of the SFI tool or a custom injector written in the C++ language.

*Action : Perturb :: built-in/custom injector*

### 4.2 Aspect Design

We design aspects (advice and point-cut) using FIDL scripts. FIDL scripts are very short, simple, and use abstract entities defined in the previous section. This allows testers to avoid dealing with the internal details of the SFI tool or the underlying compiler (LLVM in our case), and substitutes the complex fault model design process with a simple scripting process. As indicated in Fig 2, FIDL core weaves the defined aspects into LLFI source code by compiling aspects into fault triggers and fault injectors, and automatically integrating them into LLFI.

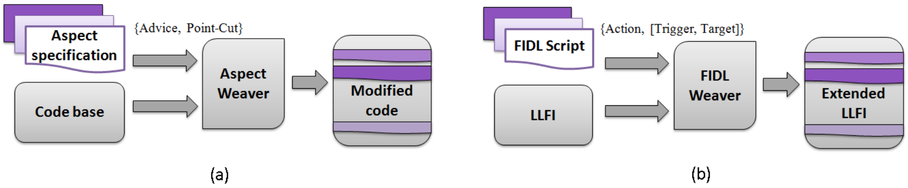

Fig. 2: (a) Aspect-oriented software development [7], (b)FIDL as an AOP-based language.

Algorithm 1 describes how FIDL designs aspects, and how it weaves the aspects into LLFI source code. For the instructions that belong to both Trigger and Trigger* sets (line 1), algorithm 1 looks for the register(s) that are defined in Target (line 2). Every pair of instruction and corresponding register provides the required information for building PointCut (line 3). FIDL takes the Action description to build Advice (line 4), that is paired with PointCut to form a FIDL aspect (line 5). Now, algorithm 1 walks through the AUT's code, and looks for the pairs of instruction and register(s) that match to those of PointCut (line 8). Then, it generates the fault trigger and fault injector's code in C++ (line 9, 10). Fault trigger is a LLVM pass that instruments the locations of code identified by PointCut during compile time, and fault injector is a C++ class that binds the Advice to the locations pointed to by PointCut during run time.

**Algorithm 1** FIDL weaver description

```
for all $inst_i \in (Trigger \cap Trigger*)$ do
    for all $reg_j \in Target$ do
        $PointCut[i, j] \leftarrow [inst_i, reg_j]$
        $Advice \leftarrow Action$
        $Aspect \leftarrow [Advice, PointCut[i, j]]$
Iterate all basic blocks of $AUT$
for all $[inst_m, reg_n] \in AUT$ do
    for all $[inst_m, reg_n] = PointCut[i, j]$ do
        $FaultTrigger_k \leftarrow PointCut[i, j]$
Generate $FaultInjector$ from $Advice$
```

## 5 Evaluation metrics

We propose three metrics for capturing the efficiency of our programmable fault injection framework, *(1)complexity, (2)time overhead, and (3)implementation overhead*. We apply these metrics to the SFI campaign that utilizes different fault models across multiple AUTs. For each metric, we compare the corresponding values in FIDL with the original fault injectors implemented in the LLFI framework (in C++ code). Before we explain the above metrics, we describe the possible outcomes of the fault injection experiment across AUTs as follows:

- *Crash*: Application is aborted due to an exception.
- *Hang*: Application fails to respond to a heartbeat.
- *SDC* (Silent Data Corruption): Outcome of application is different from the fault-free execution result (we assume that the fault-free execution is deterministic, and hence any differences are due to the fault).
- *Benign*: None of the above outcomes (observable results) with respect to either fault masking or non-triggering faults.

**Complexity** is defined as the effort needed to set up the injection campaign for a particular failure mode. Complexity is measured as time or man hours of uninterrupted work in developing a fault model. Because this is difficult to measure, we calculate instead, the number of source Lines Of Code (LOC) associated with a developed fault model [22]. We have used the above definition for measuring of both OFM's and FFM's complexities. *OFM (Original Fault Model)* is the fault model which is primarily developed as part of the LLFI framework in C++ language. *FFM (FIDL-generated Fault Model)* is the fault model which is translated from FIDL script to C++ code by the FIDL compiler (our tool).

**Time overhead** is the extra execution time needed to perform fault-free (profiling) and faulty (fault injection) runs respectively compared to the execution time of AUT within our framework. To precisely measure the average time overhead of each SFI campaign, we only include those runs whose impact are SDCs, as the times taken by Crashes and Hangs depend on the exception handling overheads and the timeout detection mechanisms respectively, both of which are outside the purview of fault injections. We also exclude the benign results in time overhead calculations, because we do not want to measure time when the fault is masked as these do not add any overhead.

**Implementation overhead** is the number of LOC introduced by the translation of the FIDL scripts into C++ code. The core of FIDL includes a FIDL compiler written in Python, and three general template files to translate FIDL scripts to respective fault trigger and fault injector modules. FIDL core's is less than 1000 (Lines of Code) LOC. However, FIDL uses general templates to generate fault models' source code, which introduces additional space overhead. To measure this overhead, for every given fault model, we compared the original LOC of OFMs and those of FIDL-generated ones.

# 6 Evaluation

## 6.1 Experimental setup

**Fault models:** Using FIDL, we implemented over 40 different fault models that had originally been implemented in LLFI as C++ code [2]. However, due to time constraints, we choose five fault models for our evaluation, namely *Buffer overflow, Memory leak, Data corruption, Wrong API and G-heartbleed* (Details in Table 1). We limited the number of applied fault models to 5, as for a given fault model, we need to perform a total of 20,000 runs (two types of campaigns (2*2000 runs) with and without FIDL, across 5 benchmarks) for obtaining statistically significant results, which takes a significant amount of time.

Table 1: Sample fault model description

| Fault model | Description |
|---|---|
| Buffer overflow | The amount of data written in a local buffer exceeds the amount of memory allocated for it, and overwrites adjacent memory. |
| Data corruption | The data is corrupted before or after processing |
| Memory leak | The allocated memory on the heap is not released though its not used further in the program. |
| Wrong API | Common mistakes in handling the program APIs responsibility for performing certain tasks such as reading/writing files. |
| G-heartbleed | A generalized model of the Heartbleed vulnerability, that is a type of buffer over-read bug happening in memcpy(), where the buffer size is maliciously enlarged and leads to information leakage [20] |

[2] Available at: *https://github.com/DependableSystemsLab/LLFI*

**Target injection:** We selected five benchmarks from three benchmark suites, SPEC [10], Parboil [19], and Parsec [2]. We also selected the Nullhttpd web server to represent server applications. Table 2 indicates the characteristics of benchmark programs. The *Src-LOC* and *IR-LOC* columns refer to the number of lines of benchmark code in C and LLVM IR format respectively. In each benchmark, we inject 2000 faults for each fault model - we have verified that this is sufficient to get tight error bars at the 95% confidence intervals.

Table 2: Characteristics of benchmark programs

| Benchmark | Suite | Description | Src-LOC | IR-LOC |
|---|---|---|---|---|
| mcf | SPEC | Solves vehicle scheduling problems planning transportation | 1960 | 5054 |
| sad | Parboil | Sum of absolute differences kernel, used in MPEG video encoder | 1243 | 3700 |
| cutcp | Parboil | Computes the short range components of Coulombic potential at grid points | 1645 | 4200 |
| blackscholes | Parsec | Option pricing with Black-Scholes Partial Differential Equations | 1198 | 3560 |
| null httpd | Nulllogic | A multi-threaded web server for Linux and Windows | 2067 | 6930 |

**Research Questions:** We address three questions in our evaluation.

***RQ1****: How much does FIDL reduce the complexity of fault models?*

***RQ2****: How much time overhead is imposed by FIDL?*

***RQ3****: How much implementation overhead is imposed by FIDL?*

### 6.2 Experimental Results

Figure 4 shows the aggregate percentage of SDCs, crashes and benign fault injections (FI) observed across benchmarks for each of the fault models. We find that there is significant variation in the results depending on the fault model.

**Complexity (RQ1):** For each of the fault models, we quantitatively measure how much FIDL reduces the complexity of fault model development in our framework. Table 3 compares LOC of original fault models primarily developed in the C++ language, and fault models described in FIDL scripts. As can be seen, the LOC of FIDL scripts is much smaller than OFM ones, e.g, 10 LOC of FIDL script against 112 LOC of C++ code for developing G-heartbleed fault model. *Thus, FIDL considerably reduces the fault model complexity by 10X, or one order of magnitude, on average, across fault models.*

**Time overhead (RQ2):** Our first goal of time overhead evaluation is measuring how much LLFI slows down AUTs' execution by itself, even without FIDL. Given an OFM, we measured the average execution time for both profiling and fault injection steps, and computed the respective time overheads (*TP* and *TF*).

Table 3: Comparing the complexity of FIDL scripts with original and FIDL-generated fault models

| Fault model | OFM (LOC) | FFM (LOC) | FIDL script (LOC) |
|---|---|---|---|
| Buffer overflow | 68 | 96 | 9 |
| Memory leak | 68 | 71 | 11 |
| Data corruption | 61 | 64 | 8 |
| Wrong API | 109 | 111 | 11 |
| G-Heartbleed | 81 | 112 | 10 |

We analyzed the results to figure out how time overhead varies for each fault model across benchmarks. We find that both *TP* and *TF* increase when the number of candidate locations for injecting the related fault increases, especially when the candidate location is inside a loop. For example, the number of memory de-allocation instances(*free()* calls) within *cutcp* and *mcf* benchmarks are 18 and 4 respectively, and as can be seen in Fig 3(c), the associated *TF* and *TP* varies between 161-196% and 59-115% for these benchmarks. In this figure, the maximum and minimum time overhead are related to the *sad* and *blackscholes* with respective maximum and minimum number of *free()* calls.

Secondly, we aim to analyze how FIDL influences the time overhead. To do so, we repeated our experiments using FIDL-generated fault models, and measured the associated time overhead across benchmarks. As shown in Fig 3, the time overhead either shows a small increase or does not change at all. We also find that there is a positive correlation between the increased amount of time overhead and the additional LOC that FFMs introduce. For example, the G-heartbleed fault model imposes the maximum increase in time overhead (6.7%), and its implementation overhead has the highest value (21 LOC).

**Implementation overhead (RQ3):** We measured FIDL-generated failure modes (*FFM*) to calculate the respective implementation overhead in terms of the additional LOC (Table 3). We find that the implementation overhead for the selected fault models varies between 3-18 percent. As mentioned earlier, we find that the associated time overhead for the respective fault model with maximum implementation overhead is 6.7% , which is negligible.

## 7 Summary

In this paper, we proposed FIDL (fault injection description language) that enables the programmability of compiler-based Software Fault Injection (SFI) tools. FIDL uses Aspect-Oriented Programming (AOP) to dynamically weave new fault models into the SFI tool's source code, thus extending it. We compared the FIDL fault models with hand-written ones (in C++) across five applications and five fault models. Our results show that FIDL significantly reduces the complexity of fault models by about 10x, while incurring 4-18% implementation overhead, which in turn increases the execution time of the injector by atmost 7% across five different programs, thus pointing to its practicality.

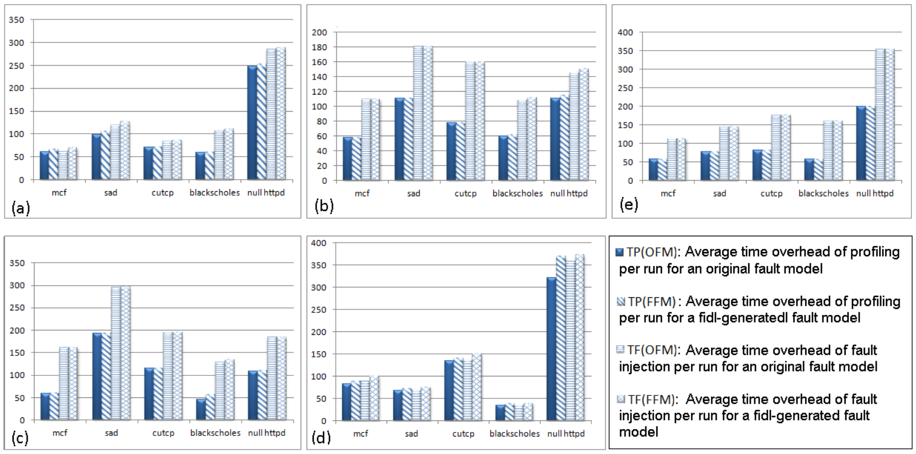

Fig. 3: Comparing Time overhead (%) of selected fault model across benchmarks; (a)buffer overflow , (b) data corruption , (c) memory leak , (d) G-heartbleed, (e) Wrong API.

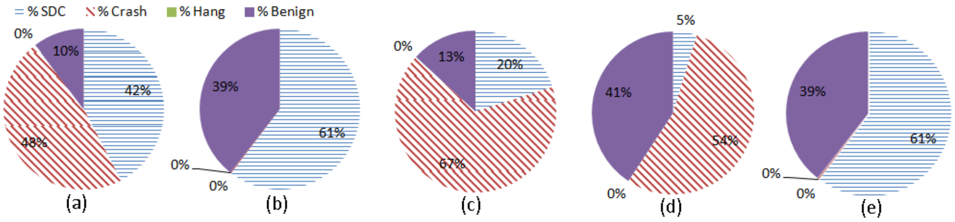

Fig. 4: Distribution(%) of aggregate impact types of sample fault models over 5 programs; (a) data corruption , (b) buffer overflow , (c) memory leak , (d) Wrong API , (e) G-heartbleed.

**Acknowledgements:** This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and a gift from Cisco Systems. We thank Nematollah Bidokhti for his valuable comments on this work.

## References

1. Maryam Raiyat Aliabadi, Karthik Pattabiraman, and Nematollah Bidokhti. Soft-LLFI: A comprehensive framework for software fault injection. In *ISSRE 2014*, pages 1–5, 2014.
2. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Parallel architectures and compilation techniques*, pages 72–81, 2008.
3. Pete Broadwell, Naveen Sastry, and Jonathan Traupman. FIG: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems*, 2002.

4. Domenico Cotroneo, Anna Lanzaro, Roberto Natella, and Ricardo Barbosa. Experimental analysis of binary-level software fault injection in complex software. In *EDCC 2012*, pages 162–172, 2012.
5. Domenico Cotroneo and Roberto Natella. Fault injection for software certification. *IEEE Transaction on Security and Privacy*, 11(4):38–45, 2013.
6. Scott Dawson, Farnam Jahanian, and Todd Mitton. Experiments on six commercial tcp implementations using a software fault injection tool. *Softw. Pract. Exper*, 27(12):1385–1410, 1997.
7. Robert Filman, Tzilla Elrad, Siobhán Clarke, et al. *Aspect-oriented software development*. Addison-Wesley Professional, 2004.
8. Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. EDFI: A dependable fault injection tool for dependability benchmarking experiments. In *PRDC 2013*, pages 31–40, 2013.
9. Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.
10. John L Henning. SPEC CPU2000: Measuring cpu performance in the new millennium. *IEEE Transactions on Computer*, 33(7):28–35, 2000.
11. Pallavi Joshi, Haryadi S Gunawi, and Koushik Sen. PREFAIL: A programmable tool for multiple-failure injection. In *ACM SIGPLAN Notices*, volume 46, pages 171–188, 2011.
12. Lukasz Juszczyk and Schahram Dustdar. Programmable fault injection testbeds for complex soa. In *Service-Oriented Computing*, pages 411–425. 2010.
13. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO 2004*, pages 75–86, 2004.
14. Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pattabiraman. LLFI: An intermediate code-level fault injection tool for hardware faults. In *QRS 2015*, pages 11–16, 2015.
15. Henrique Madeira, Diamantino Costa, and Marco Vieira. On the emulation of software faults by software fault injection. In *DSN 2000*, pages 417–426, 2000.
16. Paul D Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *DSN 2009*, pages 379–388, 2009.
17. Gail C Murphy, Robert J Walker, and ELA Banlassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on Software Engineering*, 25(4):438–455, 1999.
18. Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL: Towards a versatile fault-injection experiment framework. In *ARCS 2012*, pages 1–5, 2012.
19. John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. PARBOIL: A revised benchmark suite for scientific and commercial throughput computing. *RHPC 2012*, 2012.
20. Jun Wang, Mingyi Zhao, Qiang Zeng, Dinghao Wu, and Peng Liu. Risk assessment of buffer heartbleed over-read vulnerabilities. In *DSN 2015*, 2015.
21. Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *DSN 2014*, pages 375–382, 2014.
22. Stefan Winter, Cons Sârbu, Neeraj Suri, and Brendan Murphy. The impact of fault models on software robustness evaluations. In *ICSE 2011*, pages 51–60, 2011.
23. Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, pages 45–60, 2006.