EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments

Cristiano Giuffrida, Anton Kuijsten, Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam
{giuffrida,akuijst,ast}@cs.vu.nl

***Abstract*—Fault injection is a pivotal technique in dependability benchmarking. Unfortunately, existing general-purpose fault injection tools either inject faults in predetermined memory locations or resort to random injection, approaches that generally result in poor fault coverage and controllability guarantees. This makes it difficult to reproduce or compare experiments across different systems or workloads.**

This paper presents EDFI, a new tool for *dependable* general-purpose fault injection experiments. EDFI combines static and dynamic program instrumentation to perform *execution-driven fault injection*, a technique which allows realistic software faults to be injected in a controlled way as the target system executes. Our instrumentation strategy guarantees a predetermined faultload distribution during the entirety of the experiment, independently of the particular system or workload considered. Our evaluation confirms that EDFI significantly improves the precision and controllability of prior tools, at the cost of only modest memory and performance overhead during fault-free execution.

***Keywords*-Execution-driven fault injection, Dependability benchmarking, Basic block cloning.**

I. Introduction

As we enter the pervasive computing era, complex software systems play an increasingly important role in our everyday life. In this emerging landscape, assessing the dependability properties of a software system becomes a growing and critical concern. For dependability benchmarking purposes, researchers and practitioners have traditionally relied on software-implemented fault injection (SWIFI) tools, which provide a relatively inexpensive strategy to mimic real-world faults in a synthetic experimental setting.

In the past decades, fault injection campaigns have been applied to several categories of software, including distributed systems [1], user applications [2]–[4], operating systems [5], [6], device drivers [7]–[9], and file caches [10]. These experiments have served a number of different purposes, including: (i) analyzing (and comparing) the behavior of different systems under a given faultload [5], [6]; (ii) evaluating the effectiveness of fault-tolerance mechanisms [7]–[9]; (iii) performing high-coverage testing of error detection and recovery mechanisms [2], [11], [12].

Recent research efforts to build practical fault injection tools are largely focused on the latter scenario [3], [4], [11]. In this context, experiments are designed to surgically inject targeted faults into the system and trigger rarely executed code paths, rather than mimicking real-world software faults. The ultimate goal is typically to increase the code coverage explored during automated testing. Ironically, earlier efforts, which are instead focused on designing general-purpose fault injection tools, are, in turn, heavily affected by limited program code coverage achieved during the experiment.

In detail, the dominant approach followed by existing general-purpose tools is to inject faults into predetermined (or random) memory locations [1], [8], [10], [13]–[17], a *location-based* strategy which cannot guarantee that faults are actually "covered"—with a predetermined *faultload distribution*, i.e., characterization of fault locations and types (§ V)—at runtime. Not surprisingly, prior studies have reported fault injection campaigns with no faults activated in as many as 40% of the experiments [1], [10]. A way to address this problem is to substantially increase the number of faults injected, but at the cost of more experiments invalidated by spurious faults activated before even starting the test workload. An alternative is to surgically inject faults into hot spots stressed by the test workload [18]–[20], a strategy which, however, requires a deterministic workload and does not account for code covered only during faulty execution (e.g., error handling code).

Other approaches, in turn, periodically interrupt the system at random execution points (i.e., typically using a timer) and inject faults into the current runtime context [13], [16], [17]. This *time-based* strategy, however, biases the experiment toward hot code paths and severely limits the nature of the faults that can actually be injected at runtime, ultimately producing poorly representative software faults [21]–[23]. We believe all these shortcomings have significantly affected the "*dependability*" of existing tools, often even prompting researchers to question the validity of fault injection as a dependability benchmarking technique [24].

This paper presents EDFI, a new tool for *dependable* general-purpose fault injection experiments. Unlike all the prior tools, EDFI implements *execution-driven fault injection*, a novel technique which allows injecting a controlled and predetermined faultload distribution as the system executes at runtime. To address this challenge, EDFI relies on a combination of static and dynamic program instrumentation, which transforms the original code into *multiple heterogeneous versions* at compile time and provides the ability to interleave them in a controlled way during the experiment. This strategy allows EDFI to (i) statically inject multiple *simultaneous* [25] faults over the entire code—with a predetermined faultload distribution—to avoid coverage problems during the experiment and (ii) seamlessly switch between fault-free and faulty

execution to allow fault activation only in a user-controlled fault injection window at runtime.

EDFI's hybrid instrumentation strategy delivers *precise* (i.e., how well the tool follows the original fault model), *controllable* (i.e., how well the user can mark the beginning/end of an experiment with no spurious faults activated before/after then), and *observable* (i.e., how well the user can observe/measure the output of an experiment) fault injection experiments with negligible system impact during normal execution. Unlike all the existing tools, EDFI offers strong guarantees that a predetermined fault characterization given in input (i.e., *input faultload distribution*) will be precisely reflected in the observed output of the experiment (i.e., *output faultload distribution*) without introducing spurious faults that may compromise the validity of the results.

II. Background

Software-implemented fault injection (*SWIFI*) is a well-established technique in dependability benchmarking experiments. Its merit lies in emulating real-world faults in a synthetic setting with relatively simple tools.

SWIFI tools are typically designed to either inject generic faults into the original program or emulate error conditions at the library interfaces. The latter scenario is popular in robustness testing campaigns, which aim to analyze the behavior of the system in face of error codes returned by the libraries [3], [4] or invalid arguments supplied to library (or system) calls [6]. While important in robustness testing applications, these strategies are orthogonal to general-purpose fault injection techniques in terms of both goals and representativeness, as also demonstrated in prior work [26].

General-purpose SWIFI tools, in turn, are traditionally classified into two main categories, depending on whether fault injection is performed at *preruntime* or at *runtime*. The former approach injects faults by statically mutating the original program via compiler-based techniques [8], [27] or binary rewriting [10], [13]–[15]. Mutations can affect code or data and follow a predetermined *location-based* strategy. Locations are either user-defined or selected at random. Early approaches, such as [13], [14], corrupt the program image with hardware-like faults (e.g., bit flips). More recent approaches, such as [8], [10], [15], [22], [28], in contrast, explicitly aim at emulating realistic software faults. G-SWFIT [15], for example, injects only fault types obtained from the analysis of 668 real-world bugs found in the field.

In both cases, preruntime location-based approaches have a number of important shortcomings. First, fault activation cannot be easily guaranteed, as it is subject to code coverage induced by the test workload. Even when faults are activated, limited coverage immediately translates to very weak guarantees on the dynamic faultload distribution actually injected at runtime. Second, it is infeasible to prevent faults from being activated outside the user-controlled fault injection window, which should, however, clearly mark the boundaries of the experiment. This greatly limits the controllability of the approach. For example, the system may inadvertently crash at initialization time before even starting the test workload considered in the experiment.

Runtime location-based SWIFI strategies, such as those explored in [1], [13], [16], [17], seek to address the controllability issues of preruntime techniques. These strategies rely on hardware or software traps to interrupt the execution at predetermined (or random) memory locations and inject faults. This approach, however, is still inherently prone to the coverage problems discussed earlier. In addition, prior studies have shown that the low-level nature of these (and other) runtime techniques offers poor representativeness guarantees when emulating realistic software faults [22].

Other runtime SWIFI strategies, such as those explored in [13], [16], [17], have resorted to *time-based* fault triggers to periodically interrupt the execution (e.g., every 2 seconds) and inject faults into the current runtime context. While a potential solution to the coverage problems that plague all the other fault injection approaches, this strategy is hardly free from important shortcomings. First, the injection is heavily influenced by the workload and biased toward code paths that are executed more often during the experiment. Second, given that interruptions occur at random execution points, the nature of the faults that can effectively be injected is significantly constrained. This typically results in a weak and poorly predictable faultload distribution injected into the program at runtime. Not surprisingly, prior studies have found time-based approaches to be the least representative fault injection strategies [21]. Finally, the unpredictability of the injection events makes it really difficult to reproduce and compare the results across different experiments.

To conclude, a number of approaches have been devised to mitigate the coverage problems incurred by location-based techniques. The general idea is to profile the behavior of the system under the test workload and inject faults into hot spots with high probability of fault activation [18]–[20]. The main problem with these approaches is the inability to account for code paths only covered during faulty execution and the high sensitiveness to the workload. The latter, in particular, results in weak fault activation guarantees and also limits the reproducibility and comparability of the experiments. These issues have often emerged in prior studies. For example, the analysis presented in [19] assumes a deterministic test workload to obtain stable experimental results. DEFINE [1] reports no fault activation in as many as 40% of the experiments even with faults explicitly designed to match the test workload. Finally, the analysis presented in [21] reports a high-variance faultload distribution observed across repeated fault injection experiments with only slight variations in the workload.

In contrast to all the prior SWIFI strategies, EDFI's hybrid instrumentation approach provides a *dependable* fault injection environment, combining and outperforming existing preruntime and runtime approaches. Unlike traditional preruntime strategies, EDFI provides full controllability of the experiment, with faults only activated (and observed) within a user-controlled fault injection window. Unlike all the location-based strategies, EDFI is robust to limited coverage induced by

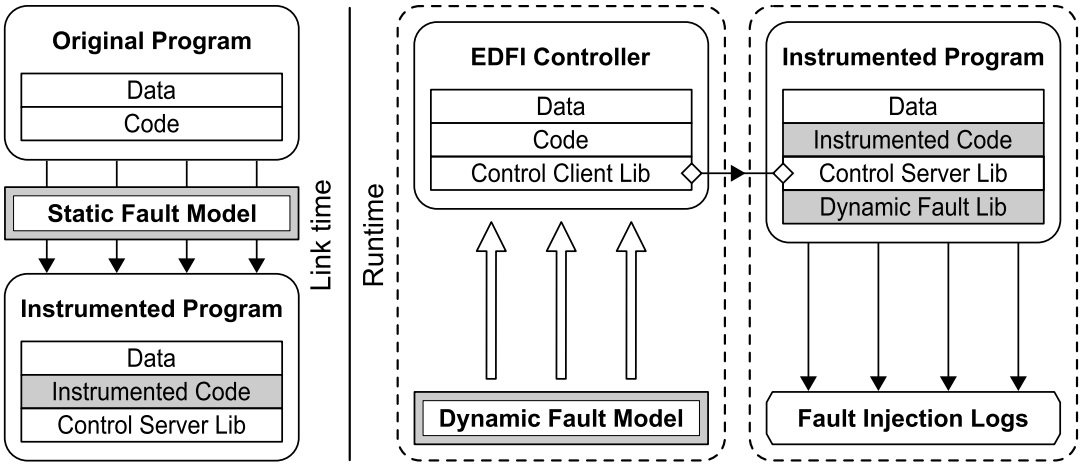


Fig. 1: Architecture of the EDFI fault injector.

the test workload. Faults are injected (and activated) directly into the currently executed code paths, independently of the particular system or workload considered. Unlike time-based strategies, EDFI imposes no restrictions on fault representativeness, nor does it yield a biased or poorly predictable fault injection experiment. Overall, EDFI's execution-driven strategy offers much stronger guarantees on the precision of the dynamic faultload, naturally yielding more reproducible and comparable fault injection results. Its LLVM-based architecture, in turn, provides a powerful, extensible, and portable framework suitable for several fault injection scenarios.

III. System Overview

The goal of EDFI is to provide a generic and extensible fault injection framework which dependability researchers and practitioners can easily adapt to their needs in many different contexts and usage scenarios. This vision is reflected in EDFI's modular and portable architecture (Figure 1).

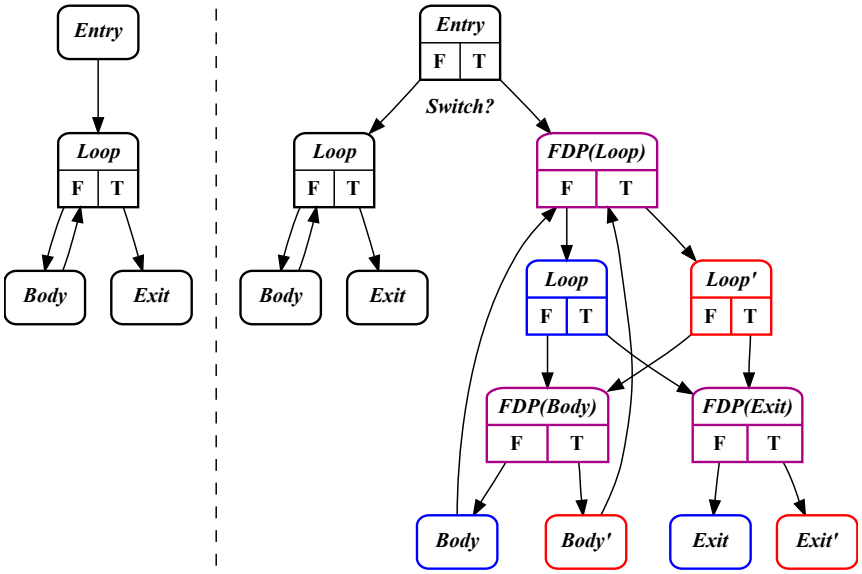
To use EDFI, users need to statically instrument the target program with a link-time transformation pass, implemented using the LLVM compiler framework [29]. The pass accepts several command-line arguments to allow the user to specify the *static fault model* (§V), which describes the input faultload distribution to inject into the program, for example, a distribution mimicking fault types found in the field [15] and locations that are representative of residual faults [28], [30]. The pass translates the static fault model requested into targeted code mutations and prepares the program for *execution-driven fault injection* (§IV). The transformations are all performed at the LLVM IR (*intermediate representation*) level before optimizations are applied. This strategy preserves the fundamental source-level abstractions required to inject realistic and representative faults [23]. In addition, the LLVM IR-level strategy seamlessly provides fault injection capabilities for all the architectures supported by LLVM.

If the source code is not available, our fault injection strategy could also be applied starting from legacy binaries. For this purpose, the LLVM MC subproject [31] includes disassemblers (with mature support for x86 and ARM) that can translate binaries into LLVM IR. Great care should, however, be taken when using this strategy, given that the resulting LLVM IR would no longer reflect the structure and the abstractions of the original source code. This issue has been also recognized in prior studies, which demonstrated the representativeness problems of binary-level fault injection [23]. In particular, the analysis in [23] found inlining and C-style preprocessor macro expansion to be the most disrupting factors for fault injection representativeness.

To avoid the representativeness problems introduced by inlined functions, our instrumentation strategy ensures that program mutations are always applied before inlining (or any other optimizations). Preprocessor macro expansion, however, is always performed in the language front end, with the original macro information irremediably lost in the LLVM IR. In its current implementation, EDFI opts for a pure LLVM IR-based strategy, losing the ability to identify the original preprocessor macros, but at the benefit of a uniform instrumentation strategy across all the source languages supported by LLVM. If macro-level representativeness is an issue in particular scenarios, a simple source-to-source transformation could be used to automatically transform function-like macros into inline functions. Recent work on source code rejuvenation demonstrates how to implement this strategy in C++11 using *perfect forwarding* [32].

A similar warning is in order for shared libraries. Our link-time transformation pass can automatically instrument the program and all the statically linked libraries. Shared libraries, however, must be separately instrumented. Nevertheless, EDFI can automatically corrupt the arguments supplied to library calls or emulate error codes returned by shared libraries, similar to the library-level strategy adopted by LFI [3].

Once instrumented, the binaries can run without deviating from their original runtime behavior. The instrumentation, however, allows the user to initiate and terminate a fault injection experiment at any point during the execution of the target system. To control the experiment at runtime, EDFI relies on two *control libraries*, which together coordinate the communication between the system and an external *controller*.



(a) Original CFG ***(b) CFG after function entry instrumentation***

Fig. 2: Basic block cloning example.

The *control server library*—which listens for external fault injection events in the background—is transparently linked against the program binary as part of our instrumentation process. The *control client library*—which delivers fault injection events to the server—provides a generic client-side interface to initialize, start, and stop a fault injection experiment on demand. EDFI already includes a simple stock *controller* (i.e., `edfi-ctl`), which relies on the control client library to expose a convenient command-line interface to the user. The client library, however, can be also as easily embedded in other complex systems to build more sophisticated controllers. Note that the control libraries are the only platform-specific components in our architecture, also designed to be easily extended and support new fault injection settings. Our current implementation includes support for UNIX applications, using UNIX domain sockets to establish the client-server communication. A portable `sysctl`-based implementation to perform fault injection into Linux and BSD OS kernels is underway.

To initiate an experiment, the user typically starts the target program, activates a test workload, and finally instructs the controller to start (and later stop) the fault injection experiment in well-known system states. To configure the experiment, the user can specify a number of parameters and a custom *dynamic fault library* (or use the stock library included in our framework), dynamically loaded into the program immediately before starting the experiment. The input to the controller defines the *dynamic fault model* (§VI) adopted, which gives the user fine-grained control over the experiment and the ability to emulate special dynamic conditions at runtime.

The user can also specify the logging mechanism to use among those supported by the control libraries. At the end of the entire process, the user can inspect the logs to determine the number, locations, types, and faultload distribution of all the faults injected during the fault injection experiment.

IV. EXECUTION-DRIVEN FAULT INJECTION

Execution-driven fault injection is a new fault injection technique which ensures predetermined faults to be systematically injected, activated, and observed as the system executes at runtime. This strategy entails several challenges. First, the faults injected during the experiment should accurately follow the faultload distribution defined by the static fault model. Second, it should be possible to seamlessly switch between faulty and fault-free execution during an experiment, as dictated by the dynamic fault model. Finally, the switching strategy should guarantee fine-grained control over the execution during the experiment, but also minimize the impact on the system during normal execution. This property is particularly important to avoid perturbing the system before initiating the experiment.

To address these challenges, our instrumentation uses the *basic block cloning* idea, which replicates and transforms the original code into multiple heterogeneous and interchangeable code versions. The general idea has been explored in prior work in different forms, using either static [33] or dynamic [34] program instrumentation strategies. EDFI embraces a new static approach to implement an efficient and flexible cloning strategy. Our transformation pass translates a generic basic block in the original control flow graph (CFG) of the program into the following basic blocks:

- **The *pristine* basic block**. This is the original basic block found in the input CFG. This block is *always* executed during normal execution at runtime when no fault injection experiment is in progress.
- **The *fault-free* basic block**. This is a semantically-equivalent copy of the *pristine* basic block, but with different predecessor and successor blocks. This basic block emulates fault-free execution within the fault injection window and can *only* be actively executed when a fault injection experiment is in progress.

```
class StaticFaultHandler {
    virtual void init(Module &M, string &params) {}
    virtual bool canInject(Value *faultLocation,
                  double faultProb) = 0;
    virtual void inject(Value *faultLocation) = 0;
};

class StaticFaultAnalyzer {
    static double getMaxSFIF(void);
    virtual void init(Module &M, string &params) {}
    virtual double getSFIF(Value *faultLocation) = 0;
};
```

Listing 1: Static fault C++ programming interface.

- **The *faulty* basic block**. This is a transformed version of the original basic block found in the input CFG, instrumented with the faultload distribution defined by the static fault model. This block emulates faulty execution within the fault injection window and can *only* be executed when an experiment is in progress.
- **The *FDP* basic block**. This is a newly generated basic block which implements the *fault decision point* (*FDP*) for the benefit of the dynamic fault model. This block determines the basic block to run next within the fault injection window, allowing the experiment to seamlessly switch between *fault-free* and *faulty* execution.

Figure 2 shows a simplified example of EDFI's basic block cloning strategy. The original CFG in the example was generated from a simple function with a single loop summing all the elements of an array. As exemplified in the figure, the transformation preserves the basic structure of the original CFG, but a number of *pristine* basic blocks are modified to check the value of a special *switch* flag and redirect execution to a different code version when necessary. To minimize the runtime overhead, the flag is only checked at function entries and loop back edges (not shown in the figure, for simplicity), similar to [33]. While efficient, this approach provides only coarse-grained control over the execution with no ability to switch to a different code version at every basic block. To address this challenge, EDFI only relies on the *switch* flag to interrupt (and restore) normal execution at the beginning (and at the end) of the fault injection window, but introduces *FDP* blocks to support basic block-level switching granularity during the experiment. Note that supporting instruction-level switching granularity is also an option, but we found this strategy to drastically increase the complexity of the CFG—hindering optimizations and encouraging memory overhead—without significantly improving the expressiveness of the dynamic fault model. If more expressiveness is necessary, our basic block cloning strategy could also be configured to generate multiple *faulty* basic block versions rather than just one for each *pristine* basic block, also providing the ability to switch between different static fault models at runtime.

At the beginning of the fault injection experiment, the control library sets the *switch* flag to allow the execution to switch to a different code version at the next function entry or loop back edge—the latter is necessary to support execution-driven fault injection in face of long-running loops. From that moment, the execution percolates through a network of *FDP* blocks, which reflects the original CFG structure but can selectively redirect the execution flow to *faulty* or *fault-free* basic block versions according to the dynamic fault model. When the *switch* flag is unset to terminate the experiment—as dictated by the dynamic fault model or by the controller—the *FDP* blocks channel the execution flow exclusively into *fault-free* basic blocks, while allowing the system to restore normal execution at the next function entry or loop back edge.

V. Static Fault Model

The goal of the static fault model is to shape the faultload distribution adopted at runtime for the fault injection experiments. In particular, the model should give the user the ability to accurately specify *what* faults to inject and with what *distribution*, according to the particular fault scenario considered. For this purpose, EDFI relies on generic *static fault handlers* (*SFHs*). A single *SFH* implements a particular fault injection strategy, characterized by a *static fault trigger* (*SFT*) (i.e., conditions that designate particular code locations for fault injection) and *fault type* (i.e., actions to inject the fault into the program). *SFHs* are implemented by pluggable objects that adhere to a well-defined C++ programming interface, shown in Listing 1.

The abstract C++ class `StaticFaultHandler` defines a number of virtual methods for the benefit of the subclasses. The optional `init()` method can be used to perform one-time initialization operations. The `inject()` method is used to implement the fault injection strategy. Finally, the `canInject()` method is used to implement the static fault trigger. Our static fault triggers are similar, in spirit, to generic fault triggers proposed in prior work [4], [35]. Our *SFTs*, however, are completely static—dynamic triggers are, in contrast, used in our dynamic fault model (§VI).

At the end of the basic block cloning process, our pass locates all the `StaticFaultHandler` implementations (built-in or user-defined) scheduled for injection according to the static fault model specified by the user. Next, the pass scans the entire LLVM IR program to identify all the candidate fault locations (e.g., store instruction). Our current implementation supports fault locations at three different levels of granularity, reflected in the `Function`, `BasicBlock`, and `Instruction` LLVM IR objects. For each candidate fault location, the pass invokes the `canInject()` method on all the designated *SFH* objects. Our current `StaticFaultHandler` implementations consider only instruction-level fault locations, but it is straightforward to implement more complex fault injection strategies that operate at the function or basic block level.

The `canInject()` method accepts two arguments: the current candidate fault location and the fault probability. The latter determines the probability that a particular fault type will be injected in a candidate fault location. The fault types (i.e., *SFHs*) to consider, their corresponding probabilities, and any other optional parameters (i.e., `params`) are specified by the user via command-line arguments to our transformation pass. These arguments reflect our definition of *faultload distribution*,

which is fundamentally different from prior characterizations adopted in the literature [15].

Traditional faultload characterizations describe the set of fault types in terms of the fraction of the total faults each fault type represents [15]. While convenient for location-based approaches and single-fault injection strategies, this definition often translates to a weak faultload characterization, which ignores the structure and size of the program. Our probability-based characterization, in contrast, is inherently code size-agnostic and enables *simultaneous* fault injection [25]. The former property is particularly important to compare fault injection results across different programs, while also giving strong guarantees that the given fault probabilities will be reflected in the output at runtime—precision problems should only be expected in cases of very limited coverage (§ VI).

As acknowledged in the analysis presented in [36], however, there are many factors that may nontrivially increase the fault density in particular code locations. For example, prior studies have shown that the fault density is statistically correlated with code complexity measures [37]. Other studies have presented empirical evidence that imports and function calls correlate with security vulnerabilities [38]. To alter the original faultload distribution and express more sophisticated fault models that consider these (and other) conditions, EDFI relies on generic *static fault impact factors* (*SFIFs*). These factors can be used to amplify or (reduce) the fault probabilities in particular code locations, orthogonally to the original fault types considered. The *SFIFs* are computed on a per-fault location basis by pluggable *static fault analyzer* (*SFAs*) objects, which also adhere to a well-defined C++ programming interface (Listing 1).

In addition to the conventional `init()` method, the abstract C++ class `StaticFaultAnalyzer` exposes two methods to the transformation pass. The virtual method `getSFIF()` returns the fault impact factor of the current candidate fault location. The static method `getMaxSFIF()` returns the maximum fault impact factor possible across all the user-specified *SFAs*. For each candidate fault location in the program, the pass invokes the `getSFIF()` method on each designated *SFA* and stops when the first valid *SFIF* for the current location is found (if any). The priority of application of a particular *SFA* is determined by the original order specified by the user. The final fault probability given to the `canInject()` method of each `StaticFaultHandler` object is the normalized version of the original fault probability, which is simply computed as:

```
faultProb *= SFA.getSFIF()/SFA.getMaxSFIF()
```

Unlike *SFIFs*, static fault triggers are never evaluated in a priority-based fashion. After calling the `canInject()` method on all the designated *SFHs*, the pass selects only those that have returned a positive answer and performs random selection to resolve eventual collisions. This strategy is necessary to avoid perturbing the faultload distribution specified by the user and also eliminate duplicate faults that can introduce representativeness problems. The selected *SFH* (if any) is finally requested to inject the fault into the program (i.e., with a call to the `inject()` method). At the end of the process, the pass reports accurate statistics on the faultload distribution injected. This is important to give the user a feedback on the quality of the final static fault model applied (e.g., a high fault collision rate may introduce discrepancies in the original faultload distribution).

EDFI includes a number of built-in *SFHs* and *SFAs* that users can combine (and extend) to express several different static fault models. In particular, EDFI implements *SFHs* for all the standard software fault types described in the literature and commonly found in the field [15], [39], [40]. In addition, EDFI can specifically emulate several memory errors, including buffer overflows, off-by-N errors, uninitialized reads, memory leaks, invalid `free()` errors, and use-after-free errors. Finally, EDFI can emulate interface errors similar to those described in [3], [6] (although we have not yet implemented LFI's return code analysis [3]), while also generalizing these strategies to generic function interfaces.

The built-in *SFAs* implemented in EDFI, in turn, can be used and combined to emulate a number of sophisticated fault injection scenarios. The most basic *SFA* (i.e., `RandomFaultAnalyzer`) allows the user to override the default fault impact factor for *N* basic blocks selected at random in the program. This strategy can be used to mimic the behavior of existing location-based fault injection strategies, as also done in our evaluation. The `FunctionFaultAnalyzer` and `ModuleFaultAnalyzer` *SFAs*, in turn, allow the user to override the fault impact factors of a set of predetermined functions or modules, respectively. This is useful, for example, to emulate and analyze the impact of particularly faulty components. Finally, the `CallerFaultAnalyzer` *SFA* allows the user to override the fault impact factors of all the instructions (or basic blocks) which call a particular set of functions. This is useful, for example, to emulate interface-level fault injection at the library interfaces [3].

Other than using the built-in *SFHs* and *SFAs*, users can easily implement their own. Using the programming interface introduced earlier, users can add new *SFHs* and *SFAs* directly to the existing framework or include them in separate LLVM plugins. The LLVM API provides several opportunities to implement complex extensions with minimal effort. For example, implementing a *SFA* that amplifies the *SFIF* according to the number of lines of code in a module or the *McCabe*'s cyclomatic complexity computed over the current function (one of the best fault predictors, according to [37]) is straightforward.

VI. Dynamic Fault Model

The static fault model describes a systematic faultload distribution for the fault injection experiment, but cannot alone express more sophisticated dynamic conditions that affect the runtime system behavior. This is the main goal of the dynamic fault model. The users specify a dynamic fault model for fault scenarios that need to alter or control the faultload distribution during the experiment at runtime. In particular, the model can be used to specify *when* to switch to faulty execution and *what* to do when faults are injected into the execution. In addition, the model defines all the actions to perform at the beginning and at the end of the fault injection experiment. To meet these

```
void edfi_onstart(edfi_context_t *context);
int  edfi_onfdp(edfi_context_t *context,
                const char *file, int line);
void edfi_onfault(edfi_context_t *context,
                  const char *file, int line,
                  int num_fault_types, ...);
void edfi_onstop(edfi_context_t *context);
```

Listing 2: Dynamic fault C programming interface.

goals, EDFI supports a convenient event-driven interface to customize and control the runtime behavior of the experiment.

In detail, EDFI's dynamic instrumentation model defines four primary events: *start event* (triggered at the beginning of the experiment, as dictated by the controller), *fdp event* (triggered at every fault decision point encountered), *fault event* (triggered when switching to faulty execution), *stop event* (triggered at the end of the experiment). For each of these events, EDFI defines a corresponding event handler in the C programming interface exported by the dynamic fault library. The four event handlers are shown in Listing 2.

Every event handler receives as an argument a pointer to the *fault injection context* (i.e., `edfi_context_t` object). The context includes all the fault injection variables that are normally used to initialize, track, and influence the state of the experiment. For example, the context holds the counters to provide statistics on the faultload distribution observed at runtime, as well as the policies to control the behavior of our stock dynamic fault library. To prevent corruption of the fault injection variables during the experiment, the control libraries guarantee that the context is always mapped high in memory and protected with guard pages.

The `edfi_onstart()` handler, automatically called when the controller signals the beginning of the experiment, initializes the fault injection context and other implementation-specific data structures. The default implementation in the stock dynamic fault library initializes the context with default values, while allowing the user to override these values through the control interface. The `edfi_onstop()` handler, automatically called at the end of the experiment, performs implementation-specific cleanup operations and outputs statistics. Our default implementation logs the termination event along with all the statistics on the faultload distribution observed. The end of the experiment can be triggered by any of the other event handlers or determined by the control libraries—in response to a user event or when a termination event is detected. To detect termination events, the current implementation of the control server library (tailored to UNIX applications) can register `atexit()` functions and abnormal termination signal handlers (e.g., `SIGSEGV`, `SIGABRT`, etc.).

The `edfi_onfdp()` handler, automatically called by our instrumentation at fault decision points, implements EDFI's *dynamic fault trigger* (*DFT*). The *DFT* returns a nonzero value to request switching to faulty execution in the next basic block. The `edfi_onfault()` handler, automatically called by our instrumentation at the beginning of a faulty basic block, implements EDFI's *dynamic fault logger* (*DFL*). The *DFL* receives as arguments the static callsite information and a variable number of arguments that indicate the types and the number of the faults injected in the current basic block. Our default implementation simply updates faultload distribution statistics in the fault injection context.

EDFI includes three main built-in *DFT* implementations:

- **Time-based *DFT***. This *DFT* can be configured to ensure a minimum predetermined time interval between faulty execution blocks. The time interval is initialized in the fault injection context and can be dynamically adjusted to specify more complex time distributions. Albeit not necessarily useful to represent realistic fault scenarios, this *DFT* can be used to analyze the behavior of existing time-based fault injection approaches.
- ***FDP*-based *DFT***. This *DFT* can be configured to ensure a minimum predetermined *FDP* interval between faulty execution blocks. This is similar to the time-based *DFT* above, but the time is measured in terms of number of *FDPs* executed instead of microseconds. This *DFT* can be used to accurately specify the timing of runtime faulty behavior in an execution-driven fashion. Unlike time-based *DFTs*, this strategy translates to reproducible and unbiased fault injection experiments.
- **Probability-based *DFT***. This *DFT* can be configured to express a predetermined dynamic probability of switching to faulty execution. As for the other *DFTs*, the probability is initialized in the context and can be dynamically adjusted to specify complex distributions. This *DFT* can be used to accurately specify the likelihood of runtime faulty execution and emulate particular fault scenarios (e.g., bug clustering effects). In addition, dynamic probabilities can be used to adjust (or replace) the static probabilities given for the static fault model in case of limited or highly polarized code coverage—which may affect the precision of the resulting output faultload distribution. For example, a program executing only a few in-loop basic blocks may result in poor precision and fault activation guarantees with particular static fault models. A possible solution is to instruct the static fault model to inject faults in every fault location candidate and rely exclusively on dynamic probabilities to shape the resulting faultload distribution.

The default *DFT* implementation in our stock dynamic fault library evaluates all the built-in *DFTs* which have been parametrized by the user (if any). Further, our default implementation allows the user to specify conditions that can automatically terminate the experiment. Termination can be triggered basing on *time*, *FDPs*, and *faults* observed from the beginning of the experiment. To parametrize the experiment, users can, for example, rely on our stock controller:

```
edfi-ctl <start|stop> [options]
```

The optional `[options]` argument allows the user to configure the fault injection context for the experiment and the dynamic fault library to use. When no option is given, EDFI resorts to the stock library implementation and systematically switches to faulty execution with no restriction during the experiment.

Other than configuring and combining the built-in *DFTs* and *DFLs*, users can easily implement their own dynamic fault library. Our C programming interface provides convenient access to the fault injection context and the entire program state. For example, it would be straightforward to implement a *DFT* that switches to faulty execution only when the program reaches a particular state, similar to [4], [35]. Further, EDFI exposes *static fault IDs* (derived by callsite information) and *dynamic fault IDs* (derived by calltrace information) directly to the *DFTs* and *DFLs*, generalizing *failure IDs* in [12]. Other than supporting simple call stack-based or call count-based triggers as in [4], this interface can be used to implement more complex dynamic fault models, including:

- **Emulate transient (or intermittent) faults**. In this scenario, the *DFL* implementation logs all the *fault IDs* in memory, allowing the *DFT* to identify duplicate *fault IDs*. To emulate transient (or intermittent) faults, the *DFT* implementation discards (or selectively enables/disables) duplicate *fault IDs* in a single run.
- **Record/replay a fault injection experiment**. In this scenario, the *DFL* implementation logs all the *fault IDs* to persistent storage. In subsequent fault injection runs, the *DFT* implementation systematically replays a previously logged run. If necessary, deterministic replay can be enforced using third-party frameworks [41].
- **Implement high-coverage fault exploration**. In this scenario, the *DFL* implementation logs all the *fault IDs* to persistent storage. The *DFT* implementation, in turn, discards duplicate *fault IDs* across different runs. More advanced fault exploration strategies, such as those in [2], [4], [11], [12] are also possible.

VII. Evaluation

Our current EDFI implementation runs on standard UNIX systems, being specifically designed to support fault injection for user-space UNIX programs. Its portability, however, is only subject to the platform-specific control libraries, which are easy to retarget given their small size (234 LOC [1]). The static instrumentation, in turn, is implemented as an LLVM pass in 1150 LOC. The stock dynamic fault library and the command-line controller, finally, are implemented in C in 259 and 55 LOC, respectively.

We evaluated EDFI on a workstation running Linux v2.6.32 and equipped with a 12-core 1.3Ghz AMD Opteron processor and 4GB of RAM. For our evaluation, we considered MySQL (v5.1.65) and Apache httpd (v2.2.23), a popular open-source DBMS and web server, respectively. To directly compare our results with recent fault injection techniques [3], we performed our tests using the SysBench OLTP benchmark [42] (MySQL) and the AB benchmark [43] (Apache httpd). We configured our programs and benchmarks using the default settings. To obtain unbiased results toward particular fault types, we allowed EDFI to use the same static fault probability $P = \Phi$ (with $\Phi = 0.5$, unless otherwise noted) in all our tests. We repeated all our experiments 21 times and report the median.

[1]Source lines of code reported by David Wheeler's SLOCCount.

Test scenario	Static HTML	PHP
Normal execution	1.024	1.007
FDPs only	1.052	1.018
Default *DFL* only	2.091	1.138
Default *DFT* (nonparametrized)	2.416	1.185
FDP-based *DFT*	4.190	1.468
Time-based *DFT*	4.206	1.472
Probability-based *DFT*	4.464	1.521

TABLE I: Time to complete the Apache benchmark (AB) normalized against the baseline.

Our evaluation answers four questions: (i) *Performance*: Does EDFI yield low runtime overhead during normal execution and reasonable slowdown during the experiment? (ii) *Memory usage*: How much memory does EDFI use? (iii) *Precision*: Does EDFI yield more precise faultload distributions than prior tools? (iv) *Controllability*: Does EDFI yield more controllable experiments than prior tools?

Performance. We evaluated the runtime overhead imposed by the fault injection mechanisms used in EDFI. To this end, we evaluated our application benchmarks in a number of test scenarios. In the first scenario, we instrumented the applications and measured the overhead imposed on our benchmarks during normal execution. The question we wish to address is whether our instrumentation introduces minimal impact on normal execution and the overhead of checking the *switch* flag is effectively amortized by hardware caches and branch prediction mechanisms. In the second scenario, we measured the overhead imposed on our benchmarks during a fault injection experiment with no *DFTs* and *DFLs* used. This scenario isolates the overhead of basic block-level switching introduced by the *FDPs*. The third and fourth scenarios, in turn, add the default *DFL* and the default nonparametrized *DFT* (respectively) to the previous configuration, isolating their overheads for comparison. Finally, the last three scenarios measure the overhead of *FDP*-based, time-based, and probability-based *DFTs*, respectively. In all the experiments, EDFI's instrumentation is configured to skip (*only*) the code mutations that inject the actual faults. This is necessary to allow our benchmarks to complete and obtain representative and unbiased performance results.

We first evaluated our test scenarios with Apache httpd, measuring the time to complete the AB benchmark compared to the baseline. Similar to [3], we ran 1,000 requests—designed to retrieve a 1 KB file—and two different workloads (static HTML and PHP) with AB in each test scenario. For the PHP workload, we did not instrument `mod_php` to evaluate the impact of uninstrumented shared libraries. Table I presents our results. As shown in the table, the overhead introduced by our instrumentation during normal execution is negligible for the two workloads (2.4% and 0.7%), directly comparable, for example, to LFI executing with 4 triggers [4]. The other test scenarios, in turn, highlight the overhead of our *FDPs*, *DFLs*,

Test scenario	Read-only	Read-write
Normal execution	1.053	1.054
FDPs only	1.095	1.060
Default *DFL* only	1.161	1.070
Default *DFT* (nonparametrized)	1.213	1.116
FDP-based *DFT*	1.509	1.408
Time-based *DFT*	5.201	3.920
Probability-based *DFT*	7.448	5.638

TABLE II: MySQL throughput normalized against the baseline.

Type	Apache httpd	MySQL
Static	1.919	1.418
Runtime (idle)	1.015	1.445
Experiment initialization	1.015	1.445
Experiment in progress	1.015	1.329

TABLE III: Memory usage normalized against the baseline.

and *DFTs* during the fault injection experiment. Compared to LFI, the overhead grows considerably when evaluating additional triggers (and event handlers in general), reaching maximum values of 346.4% and 52.1% with the stock *DFL* and probability-based *DFT* enabled. This behavior is, however, to be expected, given that LFI solely operates at the library interfaces. EDFI's execution-driven fault injection strategy, in contrast, aims at full execution coverage. While nontrivial, this overhead is strictly confined in the fault injection window and always conditioned by the complexity of the dynamic fault model adopted by the user. For example, the basic EDFI configuration with no *DFTs* and no *DFLs* reported an overhead of only 5.2% and 1.8%.

In our second run of experiments, we evaluated our test scenarios with MySQL, measuring the throughput during the execution of the SysBench OLTP benchmark compared to the baseline. Similar to [3], we ran two different workloads (read-only queries and read-write queries) in each test scenario. Table II presents our findings. As shown in the table, the results match the behavior of our earlier experiments performed on Apache httpd, with negligible performance overhead reported during normal execution (5.3% and 5.4%) and maximum performance overhead (644.8% and 463.8%) with the stock *DFL* and probability-based *DFT* enabled.

Memory usage. Our hybrid instrumentation leads to larger binary sizes and larger runtime memory footprints. This stems from our basic block cloning strategy and the libraries required to support fault injection capabilities. To evaluate their impact, we measured the memory overhead incurred by Apache httpd and MySQL when instrumented with our stock EDFI components in their default configuration. Table III presents our results. The static memory overhead (91.9% and 41.8%, respectively) measures the impact of our basic block cloning strategy and our stock control server library on the binary size. The runtime (idle) overhead (1.5% and 44.5%, respectively) measures the impact of the same instrumentation on the virtual memory size observed at runtime, right after initialization. The next row in the table is similar, but shows the virtual memory overhead at the beginning of the experiment, with our stock dynamic fault library already loaded in memory and only marginally impacting Apache httpd and MySQL's memory footprint. The last row, finally, shows the average virtual memory overhead observed within the fault injection window during the execution of our benchmarks (1.5% and 32.9%, respectively). As expected, the memory overhead introduced by EDFI is heavily influenced by the structure of the application. For example, Apache httpd reports a very low virtual memory overhead due to its large memory footprint—234MB after initialization, compared to only 42MB for MySQL. Overall, EDFI's memory overhead is modest, confirming the realistic and practical deployment of our techniques.

Precision. To assess the effectiveness of EDFI's execution-driven fault injection strategy, we evaluated the precision of the faultload distribution observed in the output of a fault injection experiment. For this purpose, we performed repeated experiments with increasing values of the fault probability Φ. In each experiment, we synchronized the fault injection window with the execution of our benchmarks, while collecting statistics on the faultload distribution observed in output. From the statistics, we directly computed the output fault probabilities for each fault type and compared their values with the input fault probabilities statically applied by our instrumentation. From the input and output probabilities collected, we computed—in each test scenario—the *faultload degradation*, which we define as the median relative error (*MRE*) across all the output fault type probabilities observed in the experiment. The faultload degradation gives an indication of the error the tool introduces when representing the output faultload distribution starting from the original input distribution specified by the user. We selected *faultload degradation* as a measure of precision, since it captures both (i) the ability of a tool to actually activate the faults specified by the user without being affected by code coverage problems and (ii) its ability to preserve the original distribution of fault types considered.

To allow our benchmarks to complete correctly, we again configured EDFI's instrumentation to skip (*only*) the code mutations that inject the actual faults. To compare EDFI's fault injection strategy with prior approaches, we also simulated location-based strategies and interface-level strategies using our built-in *SFAs*. We did not consider runtime time-based strategies in our evaluation, given that prior studies have already demonstrated their serious representativeness problems [21]. Using the `CallerFaultAnalyzer` *SFA*, we simulated interface-level strategies by instructing EDFI to inject faults only into basic blocks that contained library calls into *libc*. We specifically selected *libc* as a reference library for our experiments to obtain general and unbiased results. Using the `RandomFaultAnalyzer` *SFA*, we simulated location-based strategies by instructing EDFI to inject faults only into β basic blocks selected at random at every run (averaging the results over 201 runs). For comparability purposes, we selected the value of β according to the number of basic blocks

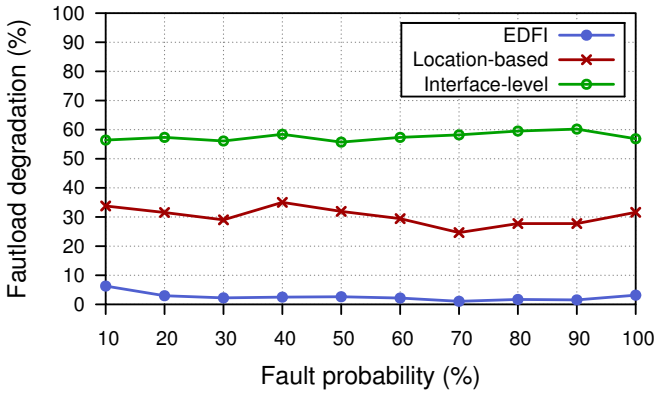


Fig. 3: Comparative faultload degradation for Apache httpd (static HTML).

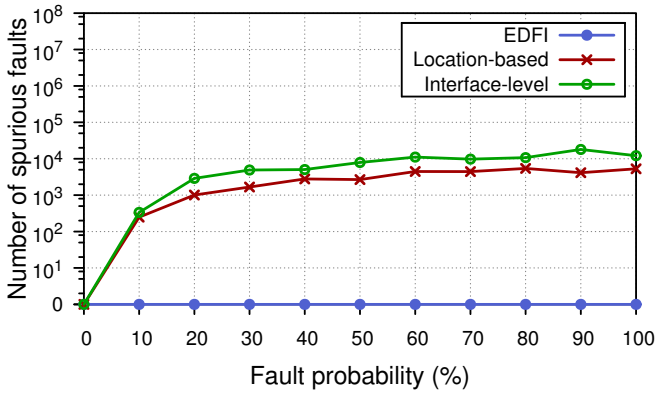


Fig. 4: Comparative number of spurious faults activated during Apache httpd initialization.

that contained at least one library call into *libc* (1689 and 1808 for Apache httpd and MySQL, respectively). Figure 3 presents our results for the Apache benchmark (AB) (static HTML). We omit the results obtained for the PHP workload and for MySQL (read-only queries and read-write queries), since they matched the behavior observed for Apache httpd (static HTML) with no significant difference.

As shown in the figure, EDFI generated a very precise faultload distribution in output, with almost no faultload degradation for any choice of the fault probability Φ. This demonstrates the benefits of injecting faults over the entirety of the program code. The other fault injection strategies, in contrast, generated imprecise faultload distributions in output, with much higher faultload degradation across all the experiments. This behavior stems from the limited coverage achieved by existing strategies. Interestingly, the location-based strategy reported lower faultload degradation (30% on average) compared to the interface-level strategy (57% on average). We interpret this behavior as the ability of random injection to achieve better coverage (on average) than injection into predetermined interface-level locations.

Controllability. We also evaluated the controllability properties of EDFI when compared to prior approaches. In particular, the question we wish to address is how well EDFI improves prior strategies in terms of user control over the fault injection experiment. For this purpose, we evaluated the number of spurious faults (i.e., faults activated before starting the experiment) introduced by the different strategies during the initialization of Apache httpd. The rationale is that every reasonable fault injection strategy should allow the target program to complete initialization before starting the fault injection experiment under a user-specified test workload. If spurious initialization-time faults are activated, however, the target program may prematurely crash (or reach a tainted and nonrepresentative state), thus compromising the validity of the entire fault injection experiment.

As done earlier, we performed repeated experiments with increasing values of the fault probability Φ. We also simulated location-based and interface-level strategies using our built-in *SFAs* and the same configuration adopted earlier. A word of warning is in order for the interpretation of the results in this particular test scenario. Our controllability analysis is only applicable to static (location-based and interface-level) fault injection strategies. Dynamic strategies—such as LFI [3]—are not affected by controllability issues, given that faults are always injected on demand and under direct control of the user. Figure 4 presents our results.

As expected, EDFI reported no spurious faults during the initialization of Apache httpd. For the other strategies, in contrast, the number of spurious faults increases with the value of the fault probability Φ. This behavior is expected, given the higher chances of fault activation in various (and arbitrary) parts of the program. As the figure shows, the interface-level strategy reported a consistently higher number of spurious faults compared to the location-based strategy, with 12120 and 5309 faults (respectively), for $\Phi = 100\%$. We interpret this behavior as a result of the particularly high density of *libc* calls during initialization.

This test scenario also highlights the precision-controllability tradeoff for existing static fault injection strategies. A larger number of faults injected results in better precision but, at the same time, lower controllability. To better investigate this tradeoff, we evaluated the impact of varying the value of the number of faulty basic blocks β in location-based strategies. This experiment was useful to understand the impact of fault coverage on static fault injection strategies, location-based approaches in particular. Figure 5 presents our findings. For low fault coverage values (e.g., around 10%), the number of spurious faults is more limited (around 7550), but faultload degradation is high (around 13%), thus resulting in poor precision. Conversely, for high fault coverage values (e.g., around 90%), faultload degradation is much lower (around 2%), but the number of spurious faults observed is substantial (around 270850), thus resulting in poor controllability. This experiment efficaciously pinpoints important limitations in existing strategies, while highlighting the better controllability and precision properties of EDFI's execution-driven fault injection strategy.

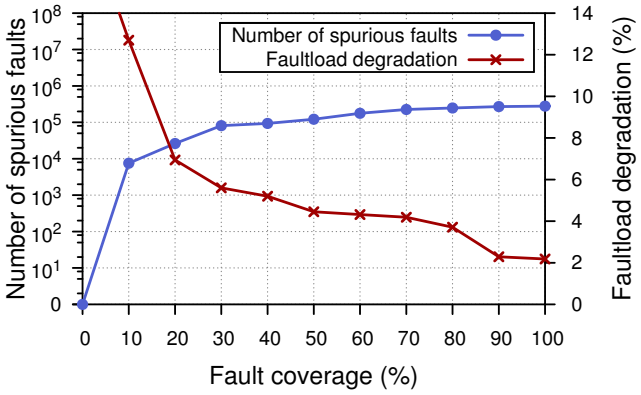


Fig. 5: Impact of fault coverage on location-based strategies (Apache httpd).

VIII. Conclusion

Fault injection experiments have been long proposed as an answer to an important question in the dependability community: "*How can we thoroughly assess the dependability of a software system?*" Undoubtedly, another equally important question is: "*How can we thoroughly assess the dependability of fault injection experiments?*" We believe the answer lies in building a new generation of general-purpose fault injection tools that can support truly precise, controllable, and observable fault injection experiments in a controlled setting. EDFI represents an important step in this direction.

EDFI injects faults in a controlled way during the execution to ensure a predetermined faultload distribution at runtime. Its hybrid instrumentation strategy provides fine-grained control over the experiment, while avoiding unnecessary perturbations to the system—or its performance—during fault-free execution. Its portable and extensible LLVM-based architecture can support several possible static and dynamic fault models, generalizing existing general-purpose fault injection tools while providing the ability to adapt to different execution contexts.

Our ultimate goal is to foster the development of a common fault injection framework for dependability researchers and practitioners, in order to support dependable, reproducible, and comparable experiments in fault injection campaigns.

Acknowledgment

We would like to thank the anonymous reviewers for their comments. This work has been supported by European Research Council under ERC Advanced Grant 2008 - R3S3.

References

[1] W.-L. Kao and R. Iyer, "DEFINE: A distributed fault injection and monitoring environment," in *Proc. of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1994, pp. 252–259.

[2] R. Banabic and G. Candea, "Fast black-box testing of system recovery code," in *Proc. of the Seventh ACM European Conf. on Computer Systems*, 2012, pp. 281–294.

[3] P. Marinescu and G. Candea, "LFI: A practical and general library-level fault injector," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2009, pp. 379–388.

[4] P. D. Marinescu, R. Banabic, and G. Candea, "An extensible technique for high-precision testing of recovery code," in *Proc. of the USENIX Annual Tech. Conf.*, 2010, pp. 23–23.

[5] W. Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Z. Yang, "Characterization of Linux kernel behavior under errors," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2003, pp. 459–468.

[6] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proc. of the 16th Int'l Symp. on Reliable Distributed Systems*, 1997, p. 72.

[7] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Trans. Comput. Syst.*, vol. 24, no. 4, pp. 333–360, 2006.

[8] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, "SafeDrive: Safe and recoverable extensions using language-based techniques," in *Proc. of the Seventh USENIX Symp. on Operating Systems Design and Implementation*, 2006, pp. 45–60.

[9] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Failure resilience for device drivers," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2007, pp. 41–50.

[10] W. T. Ng and P. M. Chen, "The design and verification of the Rio file cache," *IEEE Trans. Comput.*, vol. 50, no. 4, pp. 322–337, 2001.

[11] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: A programmable tool for multiple-failure injection," in *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, vol. 46, 2011, pp. 171–188.

[12] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A framework for cloud recovery testing," in *Proc. of the Eighth USENIX Conf. on Networked Systems Design and Implementation*, 2011, pp. 18–18.

[13] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, 1995.

[14] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using FIAT," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 575–582, 1990.

[15] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, 2006.

[16] T. K. Tsai and R. K. Iyer, "Measuring fault tolerance with the FTAPE fault injection tool," in *Proc. of the Eighth Int'l Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, 1995, pp. 26–40.

[17] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 125–136, 1998.

[18] T. K. Tsai, M.-C. Hsueh, H. Zhao, Z. Kalbarczyk, and R. K. Iyer, "Stress-based and path-based fault injection," *IEEE Trans. Comput.*, vol. 48, no. 11, pp. 1183–1201, 1999.

[19] A. Johansson, N. Suri, and B. Murphy, "On the impact of injection triggers for OS robustness evaluation," in *Proc. of the 18th Int'l Symp. on Software Reliability Engineering*, 2007, pp. 127–126.

[20] A. Johansson, N. Suri, and B. Murphy, "On the selection of error model(s) for OS robustness evaluation," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2007, pp. 502–511.

[21] M. H. J. Christmansson and M. Rimén., "An experimental comparison of fault and error injection," in *Proc. of the Ninth Int'l Symp. on Software Reliability Eng.*, 1998, p. 369.

[22] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2000, pp. 417–426.

[23] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *Proc. of the Ninth European Dependable Computing Conf.*, 2012, pp. 162–172.

[24] P. Koopman, "What's wrong with fault injection as a benchmarking tool?" in *Proc. of the Workshop on Dependability Benchmarking*, 2002, p. 31.

[25] S. Winter, M. Tretter, B. Sattler, and N. Suri, "simFI: From single to simultaneous software fault injections," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2013, pp. 1–12.

[26] R. Moraes, R. Barbosa, J. Duraes, N. Mendes, E. Martins, and H. Madeira, "Injection of faults at component interfaces and inside the

component code: Are they equivalent?" in *Proc. of the Sixth European Dependable Computing Conf.*, 2006, pp. 53–64.
[27] J. Hudak, B.-H. Suh, D. Siewiorek, and Z. Segall, "Evaluation and comparison of fault-tolerant software techniques," *IEEE Trans. Rel.*, vol. 42, no. 2, pp. 190–204, 1993.
[28] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "On fault representativeness of software fault injection," *IEEE Trans. Softw. Eng.*, vol. PP, no. 99, p. 1, 2012.
[29] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of the Int'l Symp. on Code Generation and Optimization*, 2004, p. 75.
[30] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "Representativeness analysis of injected software faults in complex software," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2010, pp. 437–446.
[31] J. Grosbach and O. Anderson, "LLVM MC in practice," in *LLVM Developers' Meeting*, 2011.
[32] A. Kumar, A. Sutton, and B. Stroustrup, "Rejuvenating C++ programs through demacrofication," in *Proc. of the 28th IEEE Int'l Conf. on Software Maintenance*, 2012.
[33] J. Wu, H. Cui, and J. Yang, "Bypassing races in live applications with execution filters," in *Proc. of the Ninth USENIX Symp. on Operating Systems Design and Implementation*, 2010, pp. 1–13.
[34] G. Portokalidis and A. D. Keromytis, "REASSURE: A self-contained mechanism for healing software using rescue points," in *Proc. of the Sixth Int'l Conf. on Advances in Information and Computer Security*, 2011, pp. 16–32.
[35] R. Chandra, R. Lefever, K. Joshi, M. Cukier, and W. Sanders, "A global-state-triggered fault injector for distributed system evaluation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 7, pp. 593–605, 2004.
[36] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *Proc. of the Int'l Symp. on Software Testing and Analysis*, 2004, pp. 86–96.
[37] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. of the 28th Int'l Conf. on Software Eng.*, 2006, pp. 452–461.
[38] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. of the 14th ACM Conf. on Computer and Commun. Security*, 2007, pp. 529–540.
[39] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing*, 1996, p. 304.
[40] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems," in *Proc. of the 22nd Int'll Symp. on Fault-Tolerant Computing*, 1992, pp. 475–484.
[41] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, "R2: An application-level kernel for record and replay," in *Proc. of the Eighth USENIX Symp. on Operating Systems Design and Implementation*, 2008, pp. 193–208.
[42] "SysBench," http://sysbench.sourceforge.net.
[43] "Apache benchmark (AB)," http://httpd.apache.org/docs/2.0/programs/ab.html.