An Extensible Technique for High-Precision Testing of Recovery Code

Paul D. Marinescu, Radu Banabic, George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Thorough testing of software systems requires ways to productively employ fault injection. We describe a technique for automatically identifying the errors exposed by shared libraries, finding good injection targets in program binaries, and producing corresponding injection scenarios. We present a framework for writing precise custom triggers that inject the desired faults—in the form of error return codes and corresponding side effects—at the boundary between shared libraries and applications.

We incorporated these ideas in the LFI tool chain [18]. With no developer assistance and no access to source code, this new version of LFI found 11 serious, previously unreported bugs in the BIND name server, the Git version control system, the MySQL database server, and the PBFT replication system. LFI achieved entirely automatically 35%-60% improvement in recovery-code coverage, without requiring any new tests. LFI can be downloaded from `http://lfi.epfl.ch`.

1 Introduction

Most software interacts with its environment through libraries, and most environmental events, including failures, are exposed to applications through the APIs of these libraries. Shared libraries, in particular, are widely used, as they encapsulate frequently used functionality. General-purpose applications frequently link to tens or even hundreds of shared libraries [18].

As a result, the reliable functioning of programs is tightly coupled to how well they handle the returns from shared libraries. While most of the APIs are well documented, they can be quite complex, and they differ from platform to platform in subtle enough ways to not be noticed during porting, but in sufficiently important ways to cause problems (e.g., `errno` values for the same `libc` call can vary between Linux, Solaris, and Mac OS X). Such corner cases are easy to miss and can lead to crashes or more subtle errors. For example, a `read()` call may not be retried after receiving an `EINTR` return code, causing the application to miss some data in its input stream.

These bugs are hard to find through input testing alone, because they are triggered by low-probability events that are typically input-independent and occur below the library layer. Yet there must be a way to ensure that programs with high reliability requirements, such as servers or embedded applications, use these libraries consistently with the libraries' true behavior. In particular, it is essential to verify that such software can correctly handle faults at or below the library layer, faults that manifest as errors returned through the shared libraries' interface.

The program/library boundary is an appealing location for injecting faults. First, it provides a well defined API where realistic faults can be injected. Second, most error recovery code can be exercised directly or indirectly via library-level fault injection. However, one must intelligently restrict the number of fault injections—an exhaustive injection campaign is infeasible, while a random one is unlikely to find bugs in a reasonable amount of time. One way to achieve this restriction is to target fault injection to precisely the program conditions that are of interest for testing, and none other.

This paper introduces a mechanism for high-precision fault injection. We extend our LFI library-level fault injector [18] with three new techniques: First, a *fault injection triggering* mechanism that allows testers to specify conditions under which a given call to a given library should be caused to fail. Triggers take the form of predicates on program state—global and local variables, call stack, etc.—which enable arbitrarily precise control over the fault injection process. Second, we present a *call site analysis* technique that aids in constructing these triggers by automatically identifying potentially buggy recovery code in program binaries, along with information on the ways in which the library calls at those sites can fail. The LFI analyzer automatically produces injection scenarios to exercise these code areas. Finally, we present an expanded *fault injection language*, which allows testers to devise sophisticated fault injection scenarios, as well as combine base triggers to form new triggers, without having to write any new code.

The rest of the paper provides an overview of LFI (§2), describes the triggering mechanism (§3), fault injection language (§4), and call site analysis (§5), after which we present details of our implementation (§6), evaluate our system (§7), survey related work (§8), and conclude (§9).

2 System Overview

We combine a library-level fault injector with three new elements that turn it into a high-precision testing tool: fault injection triggers, a fault injection language, and call site analysis. A developer would employ call site analysis to automatically identify good targets for testing, then potentially refine the generated test scenarios (written in the fault injection language), and finally provide all these to the fault injector. Below, we provide a brief overview of the injector, along with the three main contributions of this paper.

In order to make testing based on fault injection more accessible to programmers, we developed LFI [18], a tool for injecting faults at the boundary between programs and the shared libraries they use. LFI generates shim libraries to intercept library calls; based on predicates generated from user-provided configuration, they decide to either pass control to the original function or return an error value to the calling program (Figure 1).

The state of the art in testing software by injecting high-level faults consists largely of hand-coding the tests inside the product itself. For example, the MySQL server code has occasional custom code that returns specific errors when activated via a compile-time debug directives. In contrast to this approach, LFI decouples the testing from the target system's code, thus enabling automation and reuse of fault injection tests across many systems.

Using LFI involves two steps: First, a fault injection scenario is developed either by hand or using one of the automated techniques described later on. Second, the scenario is provided to the LFI controller, which conducts a suite of tests in which the described errors are introduced in the library API. The output of these tests can be used to diagnose and fix the identified bugs.

Underneath the covers, LFI uses the fault injection scenario to synthesize custom interposition libraries. The synthetic libraries have the same API as the original ones, but underneath the API they encode the fault injection logic. These libraries are shimmed between the program being tested and the original library(ies); multiple such synthetic libraries can coexist simultaneously. They intercept calls of interest, made from the program to the shared libraries, and return erroneous results that simulate faults in the libraries and the environment, as required by the scenario. The shimming of the generated libraries is system-specific: on Linux and Solaris, LFI uses the `LD_PRELOAD` mechanism to communicate
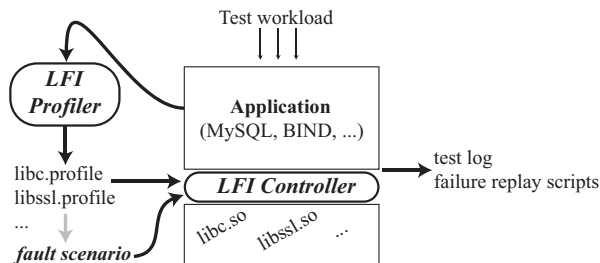

Figure 1: Architecture of the LFI fault injector.

with the dynamic linker, while on Windows it uses the Microsoft Detours framework [14].

The LFI controller coordinates the entire testing process. It is responsible for interpreting the injection scenario, generating the corresponding interception stubs, and combining them with runtime code and triggers to synthesize a new library. Once the stubs are generated and installed, the controller invokes a developer-provided script that starts the program under test, exercises it with the desired workload, and monitors its behavior to determine whether it terminates normally or with an error exit code. This information is collected in a log used by developers to diagnose and fix the program.

The LFI log records each error injection, the injected side effects (e.g., `errno`), and the events that triggered that injection (e.g., call count, stack trace). This information can be used to match injections to observed program behavior, as well as to refine the fault scenario. This also helps pinpointing and fixing the bug that caused the failure. Third party systems, like R2 [13], can be used to replay deterministically all program failures of interest.

In order to help with the generation of fault scenarios, LFI provides an automated *library profiler*, which operates directly on the binaries of the shared libraries. It performs two tasks: First, using static analysis of the binaries, it infers the return codes of the functions exported by a library (e.g., it determines that `read()` in libc can return -1, 0, or a positive number). Second, it infers side effects—besides error return values, library functions may communicate to callers additional information regarding the encountered error, via channels such as output parameters, global variables, or thread local storage (TLS) variables. For example, the profiler finds that, when returning -1, `read()` could also set the TLS variable `errno` to `EAGAIN`, `EBADF`, `EINTR`, etc. The results of these analyses are output in an XML file representing the so-called *fault profile* of the target library.

The present paper shows how we extended LFI with new techniques for writing and running sophisticated tests with little effort. Some of the key questions we address include: How to specify exactly at which point in a program's execution to inject errors? When testing a

large program, how to decide where to inject faults? Figure 2 illustrates the three new elements: a high-precision triggering mechanism, along with an expanded fault injection language and a call site analyzer.
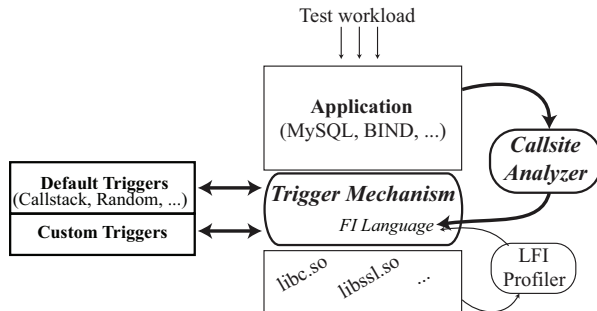

Figure 2: Architecture of the new injection framework.

A fault injection trigger is a way to specify which fault ("what") to inject at which point in a program's execution ("when") and in which call to the target library ("where"). This trigger is a predicate that evaluates to true when a fault should be injected and false otherwise. The new LFI includes default stock triggers as well as an API for writing custom triggers.

The fault injection language glues together triggers, library functions, and their fault profiles into complete fault injection scenarios. Every function which appears in a scenario is intercepted by the LFI runtime, and the associated triggers are called to decide (based on various conditions) whether to inject a fault from the profile.

The call site analyzer uses a heuristic method to check whether all error return values are checked by the callers of library functions. In other words, it searches the target program for "interesting" places to inject faults. For each identified call site, it uses dataflow analysis to determine against which error code values the return is checked. An unchecked error value indicates a potential bug, to be verified through testing. The analyzer runs on the binary, so source code of the target program is not required.

Next we describe the fault injection triggers (§3), fault injection language (§4), and call site analysis (§5).

3 Fault Injection Triggers

Triggers are invoked by the LFI runtime to decide whether an intercepted library function should fail or not. A trigger can inspect any part of system state to make its decision. Triggers offer high precision and flexibility to testers in choosing the exact conditions under which a fault is to be injected. Testers can use stock triggers with specific parameter values, they can customize existing trigger code, or write new triggers from scratch.

3.1 Trigger Interface

Triggers are pluggable modules, written as C++ classes that implement the following `Trigger` interface:

```
class Trigger {
   virtual void Init(xmlNodePtr initData) {}
   virtual bool Eval(const string&
                         libFuncName, ...) = 0;
}
```

To add a new trigger to the framework, one writes a class derived from this abstract base class and places the corresponding source files in an LFI-specific location. The new trigger can be referenced directly by class name from any injection scenario provided to LFI. We used a variant of the Registry design pattern and the Standard Template Library to implement this behavior transparently for trigger writers.

The boilerplate code needed for a trigger is minimal— usually less than 100 lines of code are needed to write a useful custom trigger. One could forgo the interface and implement triggers inside the LFI runtime, as we did in a first prototype, but then they become hard to extend and require intimate knowledge of LFI internals.

The `Init` method is optional and its default implementation is empty. It is called by the runtime after a trigger instance is created and before its `Eval` function is called for the first time. The main purpose of the `Init` function is to provide support for trigger parametrization, as we will show in §4.1. The trigger's parameters are provided as an `XMLNodePtr` object, which can be processed with a standard XML library.

The `Eval` method is where the main trigger logic resides. It is called every time a function (associated with an instance of this trigger by the injection scenario) is intercepted. Its return value indicates to the runtime whether to inject a fault or not. Since `Eval` can be called quite frequently, its code must be efficient.

`Eval` is a variadic function in order to be capable of receiving the original arguments of any intercepted library call. Its first argument indicates the name of the intercepted function; based on this name, the trigger decides how many actual arguments to expect and what their meaning is. The number of arguments must be explicitly specified in the injection scenario; since LFI does not look at source code or documentation, it cannot automatically infer the number of arguments to pass. It is possible, though, to extend LFI with LibTrac's heuristic technique for inferring function arguments [5].

In addition to the arguments passed to `Eval`, a trigger can directly obtain any other information normally accessible to a program. For example, it can use the GNU libc `backtrace()` function to inspect the call stack and determine whether the intercepted library call was made by a program, by a function in the intercepted library, or by some other shared library.

Below we illustrate trigger construction with a sketch of one of our custom triggers. It is used in an injection scenario where errors are to be injected in `read` whenever the corresponding file descriptor is a pipe, the number of bytes to be read is between 1 KB and 4 KB, and the calling thread holds a POSIX mutex.

```
#include "Trigger.h"

DECLARE_TRIGGER ( ReadPipe1K4KwithMutex )
{
private:
  static __thread int lockCount;
public:
  ReadPipe1K4KwithMutex() { }

  bool Eval(const string& libFuncName, ...) {
    pthread_t self = pthread_self();
    if (libFuncName ==
        "pthread_mutex_lock") {
      ++lockCount;
    } else if (libFuncName ==
        "pthread_mutex_unlock") {
      --lockCount;
    } else if (libFuncName ==
        "read") {
      if (lockCount > 0) {
        va_list ap;
        int fd;
        size_t size;
        struct stat st;
        va_start(ap, libFuncName);
        fd = va_arg(ap, int);
        va_arg(ap, void*);
        size = va_arg(ap, size_t);
        va_end(ap);
        fstat(fd, &st);
        return (S_ISFIFO(st.st_mode) &&
          size >= 1024 && size <= 4096);
      }
    }
    return false;
  }
};
```

`Trigger.h` contains all the required definitions for the trigger mechanism, including the definition of the `Trigger` interface. `DECLARE_TRIGGER` is a macro that simplifies the class declaration by automatically creating a properly derived class with the supplied name.

Triggers are not pure functions, but can also maintain state to inform their injection decisions. The `ReadPipe1K4KwithMutex` trigger, for instance, checks the name of the intercepted function and either updates the lock holding status for the current thread (in the case of a POSIX threads library call) or checks whether a fault should be injected in the case of the `read` function.

3.2 Stock Triggers

In addition to the `Trigger` interface, which allows testers to write their own custom triggers, LFI also provides a set of triggers that can be used out-of-the-box. We found this set to offer sufficient precision for most injection testing. Most of these triggers are generic, in that they can be used for any intercepted function and do not rely on the function's arguments. Stock triggers can easily be extended and specialized, as needed. LFI provides by default the following six triggers:

Call stack-based triggers allow injecting faults based on whether the current call stack (or part of it) matches a user-defined set of stack frames. By looking at the call stack, the trigger can learn from which particular location in the program the call is being made, from which program module (e.g., from Apache's SSL module), etc. User-provided stack frames can be identified by object file name, offsets within the binary, file name/line number pairs, or combinations thereof. The LFI call site analyzer automatically produces fault injection scenarios that use call stack-based triggers to inject faults in the locations where error checking is incomplete.

Program state-based triggers inject faults depending on whether a given relationship between program variables holds (e.g., *numConnections==maxConnections*). The stock trigger allows checking for equality between both local and global variables, but it can be easily extended with other operators. Later in the paper, we show how to specialize this trigger for data structures specific to the Apache Web server.

Call count-based triggers allow specifying that an injection should occur exactly on the *n*-th call to a function. Besides their obvious use, such triggers can also be used during debugging to replay observed failures in programs that are driven deterministically by interactions with the environment.

Singleton triggers inject a fault exactly once. This type of trigger is often combined with other triggers in a conjunction. For example, combined with a program state-based trigger, a singleton trigger can ensure that a fault is injected only the first time *numConnections==maxConnections* holds, but not afterwards. Trigger composition is described in more detail in §4.2.

Random triggers inject a fault with a configurable probability. These triggers can also be augmented with supplementary conditions, through composition.

Distributed triggers are used for testing distributed systems. A central controller receives information on intercepted calls (function name, arguments, and stack) and, based on a global view of the system, decides whether the remote trigger should fire or not. This allows setting up distributed failure scenarios, such as the ones we used for PBFT (see §7.1). In order to minimize runtime overhead, distributed triggers must be carefully composed with node-local triggers, so that the central controller is invoked solely when the injection decision can no longer be made locally.

In theory, one should write triggers that achieve perfect precision, i.e., they decide to inject a fault only in

the specific situation targeted by the tester. However, in our experience, such high precision is not always ideal: it takes longer to write an ultra-precise trigger, and the induced runtime overhead can become non-negligible. In most cases, we favor an approach where triggers are *precise enough*, i.e., inject in all targeted situations and perhaps have a couple of false positives. In the context of testing, the extra unintended fault instances might even turn out to be useful, at little extra cost.

Care must be taken to not inject *unrealistic faults*, because these can result in wasted debugging time. For example, injecting an error in an I/O call made with a blocking file descriptor and setting `errno` to `EAGAIN` is arguably unnecessarily paranoid testing, given that `EAGAIN` should only occur on non-blocking file descriptors. In LFI, this exception could be handled by composing with a trigger that evaluates to true only when the file descriptor supplied to the I/O call is non-blocking (e.g., the trigger can check the file descriptor with `fcntl`).

4   Fault Injection Scenarios

An LFI test scenario is a declaration of triggers and conditions under which these triggers should be evaluated, i.e., it is a specification of what, when and where to inject. We use an XML-based test specification language (§4.1) to describe these scenarios, including various compositions of triggers (§4.2), which permit productive reuse of fault injection scenarios. LFI employs several optimizations in the evaluation of triggers, aimed at reducing runtime overhead (§4.3).

4.1   Description Language

Fault injection scenarios can be written by hand, but we believe practitioners also want to use automated tools for generating and modifying these scenarios (such as the call site analyzer described in §5). For scenarios to be both human-readable and machine-readable, we chose an XML-based language. Here we provide an overview of the language, and direct the interested reader to the complete DTD available at `http://lfi.epfl.ch`.

An injection scenario has two main constructs: trigger declarations and associations between trigger instances and intercepted library functions. A trigger declaration makes a trigger class known to LFI and creates a named trigger instance. An association links a trigger instance to a library function that LFI intercepts, and specifies the conditions under which the triggers should be evaluated.

Consider the earlier example, which aimed to inject faults in `read` only when the corresponding file descriptor is a pipe, the number of bytes requested is between 1 KB and 4 KB, and a mutex is held by the calling thread.

The `ReadPipe1K4KwithMutex` defined in §3.1 can be associated with the relevant library calls as follows:

```
<!-- Make the trigger known to LFI -->
<trigger id="readTrig1"
         class="ReadPipe1K4KwithMutex" />

<!-- Invoke the trigger for read() calls -->
<function name="read" argc="3"
          return="-1" errno="EINVAL">
  <reftrigger ref="readTrig1" />
</function>

<!-- Trigger needs to see the lock/unlock calls -->
<function name="pthread_mutex_lock"
          return="unused" errno="unused">
  <reftrigger ref="readTrig1" />
</function>

<function name="pthread_mutex_unlock"
          return="unused" errno="unused">
  <reftrigger ref="readTrig1" />
</function>
```

The `<trigger>` element declares a trigger *instance* identified by the name `readTrig1` and implemented by the class `ReadPipe1K4KwithMutex`. This must either be a C++ class written by the tester, or an LFI stock trigger. The same trigger class can be used for multiple trigger instances.

The `<function>` element creates an association between the `read` library function and the `readTrig1` trigger instance. Whenever `read` is called by the target program, `readTrig1` is asked for a yes/no answer regarding whether to inject a fault or not. To make this decision, `readTrig1` is given by LFI three arguments from the original call stack (`argc` attribute); these arguments correspond to the file descriptor, buffer pointer, and number-of-bytes parameters of the intercepted `read` call. The trigger uses the values of these arguments to determine whether the file descriptor is a pipe and whether the requested number of bytes falls in the target range. If `readTrig1` returns true, then LFI returns to the caller a return value of -1 (`return` attribute) from `read` and sets the `errno` variable to `EINVAL`.

The other two `<function>` associations serve the purpose of informing the trigger about the corresponding mutex lock/unlock calls, giving the trigger instance the opportunity to update its state. Since these associations will never result in injections, the `return` and `errno` attributes are set to "unused."

Triggers can also be parametrized, i.e., the test scenario can specify arguments to be passed to the trigger instance at initialization time. This means, for instance, that one could replace the `ReadPipe1K4KwithMutex` class with a new class that takes the upper and lower bound of the number of bytes as arguments, instead of them being hardcoded to 1 KB and 4 KB. An example of such a class is the `ReadPipe` trigger class in the next subsection.

4.2 Trigger Composition

Triggers can be composed, to form more complex triggers. By default, associating multiple triggers within one `<function>` declaration implies a conjunction of the triggers: they all have to evaluate to true in order for a fault to be injected.

Consider the earlier pipe `read` example: instead of using the `ReadPipe1K4KwithMutex` class, we can use a conjunction of two trigger classes, `ReadPipe` and `WithMutex`. The first one handles injections in the `read` function when the file descriptor is a pipe and the number of bytes requested is between a configurable minimum and maximum. The second one injects a fault in any function, as long as the caller holds a mutex.

Trigger composition allows wider reuse of triggers and, together with parametrization, encourages writing flexible, general triggers. The scenario below illustrates the composition of two triggers, `readTrig2` and `mutexTrig`. The first `<trigger>` declaration illustrates the initialization of the parametrized `ReadPipe` trigger: it allows the tester to specify the upper and lower bound on the number of bytes, and these values are passed to the `Eval` method of the `ReadPipe` trigger.

```
<!-- Declare & initialize a ReadPipe instance -->
<trigger id="readTrig2" class="ReadPipe">
  <args>
    <low>1024</low>
    <high>4096</high>
  </args>
</trigger>

<!-- Declare a WithMutex instance -->
<trigger id="mutexTrig" class="WithMutex" />

<!-- Invoke the composition for read() calls -->
<function name="read" argc="3"
        return="-1" errno="EINVAL">
  <reftrigger ref="readTrig2" />
  <reftrigger ref="mutexTrig" />
</function>

<!-- Trigger needs to see the lock/unlock calls -->
<function name="pthread_mutex_lock"
        return="unused" errno="unused">
  <reftrigger ref="mutexTrig" />
</function>

<function name="pthread_mutex_unlock"
        return="unused" errno="unused">
  <reftrigger ref="mutexTrig" />
</function>
```

Triggers can also be composed into a disjunction, whereby a fault is injected whenever any trigger in the composition returns true; for this, one just adds multiple `<function>` elements using the same function `name`, each one associated with a different trigger. Besides conjunctions and disjunctions, LFI can support negation, whereby the result of a trigger is simply inverted. Using disjunction, conjunction, and negation, one can assemble a wide range of trigger combinations.

4.3 Trigger Evaluation

LFI evaluates triggers in the order in which they appear in the injection scenario. However, in the case of trigger compositions, LFI invokes the smallest number of triggers needed to determine the result of the composition. For example, in the case of conjunctions (i.e., multiple trigger instances referenced in the same `<function>` element), if the first trigger returns false, then the remaining triggers are not invoked at all. This optimization reduces runtime overhead and is similar to the short-circuit evaluation used in C/C++ logical expressions. This feature can be leveraged when composing with the stock singleton trigger: if added to the end of a conjunction, it ensures that a fault is injected once when all the other triggers in the composition return true.

LFI's internal data structures ensure that the list of triggers for the currently intercepted function is obtained in $O(1)$ time, i.e., it is independent of the size of the fault injection scenario. To eliminate runtime overhead during program startup, LFI uses lazy initialization: each trigger is initialized right before it is invoked for the first time.

5 Call Site Analysis

The call site analyzer helps developers find "interesting" places to inject faults, i.e., parts of the target system that are likely to be buggy. The analyzer runs entirely autonomously and looks for call sites where the return values or error side effects of the call are not properly checked. An example is the following code snippet:

```
dir = opendir(pathToDir);
while (ent = readdir(dir)) {
...
}
```

The code works properly most of the time, when `pathToDir` points to an existing directory, but crashes if the directory does not exist or `opendir` cannot allocate sufficient memory. Since the return value of `opendir` is not checked, `readdir` could be passed a null pointer. Although a rather obvious bug, we found similar bugs in widely used software like BIND and Git.

The call site analyzer combs the target program binary for places where a library function is called, and then uses dataflow analysis to determine whether the program checks for all possible errors that the function could return (as indicated by the corresponding library's fault profile, described in §2). The analyzer's method is heuristic, but we found it to be highly accurate, even if not perfect (an occasional false positive is acceptable, given that the potential bug sites can easily be verified through fault injection). While, in theory, disassembling binaries cannot be completely accurate, it has been shown that in commercial-grade applications over 99% disassembly accuracy can be achieved [22].

Once the analysis is complete, the analyzer generates a fault injection scenario targeted at the vulnerable sites with the missing errors; these scenarios employ the generic call stack trigger. The tester would then run a test based on the injection scenarios. The workload for exercising the specific call must be provided by the tester; LFI can help, by analyzing code coverage information from previous executions and indicating whether previously used workloads exercised the target call sites.

Algorithm 1 describes a simplified version of the general workflow for the analysis. The algorithm takes in the executable X, the function of interest F (e.g., `read`), and the set of error codes E, based on the library fault profile. It produces three sets of call sites: the set C_{yes} of sites where the return of F is checked for all error codes, C_{part} where it is checked for only some of the errors in E, and C_{not} where is is not checked for any errors in E.

Lines 1-2 initialize these sets and find the set $callSites_F$ of places in the target binary where F is called.

For each such call site (line 3), we construct a partial control flow graph for the instructions that *follow* the call to F (line 4), in order to determine how the return value and side effects are handled. We empirically found 100 post-call instructions to be sufficient for building the CFG we require. Indirect branches can make the CFG inaccurate, and the current LFI prototype ignores them. This is not a major issue: our analysis of over 9,000 library calls in real-world software revealed that only 0.13% (104 out of 78,292) were indirect branches.

We then perform dataflow analysis (line 5), to follow the propagation of the function's return value through the program. We look at all targets to which the return value is copied and look at all literals to which this return value (or a copy of it) is compared. We iterate through any loops that may occur, as long as the set of copies of the return value increases. In practice, this set typically stabilizes after a few iterations. Our dataflow analysis is intra-procedural; even though other functions may be called to handle errors, the real systems we analyzed always did this after a local check for error conditions.

The result of the dataflow analysis for each call to F consists of sets Chk_{eq} and Chk_{ineq}, corresponding to error codes checked via equality (as in `if (retval==-1)`) and those that are checked via inequality (as in `if (retval<0)`). If all error codes in E are checked by equality, then the call site goes into the set of fully checked calls (lines 6-7). If error codes are checked via inequality, we assume the entire range of error codes is covered (hence the disjunction in line 6). If only some of the error codes in E are checked by equality, then the call site goes into the set of partially checked calls (lines 8-9). If no error codes in E are checked, the site goes into the set of completely unchecked calls, even if error codes outside E are checked (lines 10-11).

Algorithm 1: Call site analysis (simplified)

Input: Target executable X, target function name F, target function error codes E

Output: Set C_{yes} of fully checked calls,
Set C_{part} of partially checked calls,
Set C_{not} of completely unchecked calls

1 $C_{yes} := C_{part} := C_{not} := \emptyset$
2 $callSites_F :=$ parse all calls to F in X
3 **foreach** $site \in callSites_F$ **do**
4      $cfg :=$ construct partial CFG after *site*
5      $<Chk_{eq}, Chk_{ineq}>:=$ dataflow analysis on cfg
6   **if** $Chk_{eq} \supseteq E \vee Chk_{ineq} \neq \emptyset$ **then**
7         $C_{yes} := C_{yes} \cup \{site\}$
8   **else if** $Chk_{eq} \neq \emptyset \wedge Chk_{eq} \subset E$ **then**
9         $C_{part} := C_{part} \cup \{site\}$
10   **else**
11        $C_{not} := C_{not} \cup \{site\}$
12 **return** $<C_{yes}, C_{part}, C_{not}>$

Due to space constraints, we omit the side-effect analysis done by LFI, which is responsible for verifying whether side effects shown in the fault profile (such as the `errno` variable) are properly checked. The analysis for `errno` checking is virtually identical to the one used for return values. Failing to check particular values of `errno` (e.g., not restarting a system call interrupted with `EINTR`) can compromise the application's robustness.

The call site analyzer produces two sets of fault injection scenarios, one for C_{not} and one for C_{part}. Testers are probably most interested in the former, but after exhausting the bug vulnerabilities related to C_{not}, they can make use of the latter as well. Note that the call site analyzer does not check the correctness of the error handling code, it just relieves humans of some of the burden involved in testing.

6 Implementation

At the heart of LFI is a library call interception mechanism described in more detail in the original LFI paper [18]. LFI creates a shim library that exports stub functions with the same name as the ones being intercepted. On UNIX platforms, we take advantage of the dynamic linker (using the `LD_PRELOAD` mechanism), and on Windows we use Microsoft Detours [14].

A stub function determines the address of the original function and evaluates the triggers provided in the fault injection scenario. If an injection is to be done, the stub gets the return value and side effect to be injected from the injection scenario and injects them. If no injection is to be done, the stub cleans up the stack and jumps to the original function. A stub looks approximately as follows:

```
int LIB_FUNC_NAME(void) {
  static void * (*original_fn_ptr)();
  if (!original_fn_ptr)
    original_fn_ptr = (void* (*)())
              dlsym(RTLD_NEXT, LIB_FUNC_NAME);
  if (eval_triggers(LIB_FUNC_NAME_triggers,
                    lib_function_args)) {
    /* determine return_code, side_effects */
    /* apply side_effects */
    return return_code;
  } else {
    /* return stack and registers to original values */
    __asm__("jmp [original_fn_ptr]");
    /* original function will return to caller */
  }
}
```

Since LFI has no access to source code or documentation to get the prototypes of intercepted functions, the stub functions do not have any arguments. When calling the original function (i.e., no fault injected), the stub merely removes the current frame from the stack (i.e., the one corresponding to the stub) and passes control directly to the original. This has the advantage of not requiring any information about the number of arguments and their type. When having to pass arguments to a trigger, LFI relies on the injection scenario to specify how many arguments are expected on the stack. In our current prototype, we assume all arguments are word-sized.

Designing an extensible trigger system was difficult. Our goal was to allow developers to simply drop the trigger classes in a particular location and then be able to refer to the triggers by their class name in injection scenarios. In other words, we wanted a mechanism similar to Java's `Class.forName()`. We used a variation of the Registry pattern, where each trigger class automatically inherits a factory method and a static member variable whose initialization causes the class name along with the associated factory method to be added to a global map. Instantiating a trigger is done by searching in the map and using the corresponding factory method.

To maximize ease of use, we added to LFI the ability to directly handle DWARF debug information [17]. For example, the call stack trigger allows testers to specify that injections should be done only if execution passes through certain filename/linenumber locations. Another example is the call site analyzer, which can provide the exact source code location of a particularly "suspicious" call, whenever debug symbols are available.

7 Evaluation

LFI's main strength is precision—it allows testers to specify exactly what fault to inject, where to do so, and when to inject. This can be used to selectively inject faults on a particular call when servicing a specific workload, when the program enters a particular state, or when control flow passes through a specific set of points. It is fairly obvious how, combined with knowledge of the system being tested, LFI can be a tester's "power tool."

However, such knowledge may not always available, so we focus our evaluation on how LFI can be used productively even without knowledge of the code. We inject faults in several real systems (§7.1) and find several previously unknown bugs; we also measure the improvement in recovery-code coverage. We measure the accuracy and efficiency of automated injection site identification (§7.2). We then show how LFI can be used to study the behavior of distributed systems and find interesting vulnerabilities (§7.3). Finally, we measure the overhead introduced by LFI triggers and find that their interference with the tested system is negligible (§7.4).

We evaluated LFI on four systems representing four different classes of applications: BIND 9.6.1, MySQL 5.1.44, Git 1.6.5.4 and PBFT 2008-12-09. BIND is perhaps the most popular Domain Name System (DNS) server used in the Internet, being the de facto standard for most UNIX-based network infrastructures. Git is a modern distributed version control system that was initially designed and developed for Linux kernel development, and has experienced tremendous popularity since then. MySQL is a well known and widely used open-source database management system. PBFT [7] is a practical replicated Byzantine fault tolerance system, designed to correctly serve requests in the face of f Byzantine replica failures, as long as there are at least $3f + 1$ total replicas. All experiments were run on a 4-core 2 GHz Intel Xeon CPU with 4 GB of RAM, running Ubuntu 9.04 with Linux kernel 2.6.28.

We used the binary distributions of the systems listed above. We resorted to source code only when needed to manually confirm LFI's results.

7.1 Effectiveness of Testing: Bugs and Coverage

When assessing an automated testing tool, there are generally two measures of interest: how many high-impact bugs it finds, and to what extent it improves code coverage. For this section, we run the call site analyzer on the target binaries and directly apply, with no modifications, the injection scenario it generates. Of course, the deeper the knowledge one has of the system being tested, the more effectively LFI's injection triggers can be used. However, here we focus mainly on what can be done entirely automatically, using only stock triggers. We briefly show how human intervention is useful in connecting injected faults to bugs and in writing custom triggers for particular bug categories.

As a first measure of effectiveness, Table 1 lists the 11 previously unknown bugs found by LFI entirely on its own. We expect that, in the hands of a human tester, LFI could find substantially more bugs.

We use the last bug in Table 1 to illustrate the pro-

System	Bug
BIND	***Crash*** if call to `xmlNewTextWriterDoc` fails while a user is retrieving statistics via HTTP [4]
BIND	***Abort*** due to incorrectly handled `malloc` return value in method `dst_lib_init` [3]
MySQL	***Abort*** after a double mutex unlock, due to a failed `close` [19]
MySQL	***Crash*** due to a failed `read` (error code `EIO`) while processing `errmsg.sys` [20]
Git	***Data loss*** caused by running an external command with an incomplete environment, due to failed `setenv` [11]
Git	***Crash*** due to calling `readdir` with a NULL pointer returned by a previously failed `opendir` call [9]
Git	***Crash*** due to unhandled `malloc` return value on line 567 in `xdiff/xmerge.c` [10]
Git	***Crash*** due to unhandled `malloc` return value on line 571 in `xdiff/xmerge.c` [10]
Git	***Crash*** due to unhandled `malloc` return value on line 191 in `xdiff/xpatience.c` [10]
PBFT	***Crash*** caused by a failed `recvfrom` call
PBFT	***Crash*** due to calling `fwrite` with a NULL pointer returned by a previously failed `fopen` call (see below for details)

Table 1: Bugs found automatically by LFI (more details can be found in the referenced bug reports).

cess followed in these experiments. After running on the PBFT binary, the call site analyzer generates an injection scenario, of which we show a fragment below. We then passed this scenario to the LFI injector. Upon inspecting the report of the test, we found that a replica had crashed due to a segmentation fault; the log indicated the id of the trigger that fired in that particular test case. Based on the trigger and an inspection of the source code, we immediately found that the replica's shutdown code attempts to write a checkpoint to a file, without checking that the file has been properly opened.

```
<trigger id="8054a69" class="CallStackTrigger">
   <args>
   <frame>
     <module>
       bft/bft-simple/simple-server
     </module>
     <offset>
       8054a69
     </offset>
   </frame>
   </args>
</trigger>
<function name="fopen"
   retval="0" errno="EINVAL">
   <reftrigger ref="8054a69" />
</function>
```

The other PBFT bug is interesting, as it does not manifest in the debug build, but only in the release build.

Faults were injected in `sendto` and `recvfrom` (simulating deteriorated network conditions), successively in calls made by different replicas (i.e., inject in a call made by replica R_1, then in a call made by replica R_2, etc.). This type of network behavior results in a segmentation fault in the view change phase of PBFT, when the replica tries to access a previously committed message. The reason this bug does not manifest in the debug build is because, when the debug flag is on, the PBFT implementation checks to see if messages were sent correctly and halts with an error code as soon as a problem occurs. The release (i.e., non-debug) build skips this check.

The `malloc` bug in BIND and the `close` bug in MySQL represent interesting cases of buggy recovery code. In BIND, the `dst_lib_init` method checks the return value of `malloc` calls, and runs recovery code if any such call fails. The recovery code destroys the created data structures, by calling `dst_lib_destroy`. The first statement in this method is an assertion, checking that the `dst` data structures have been initialized. However, the call from `dst_lib_init` is made before the `dst_initialized` flag is set, therefore triggering the assertion. In MySQL, the `mi_create` method has error handling code that releases resources, including a particular mutex. However, a failed `close` call can trigger this code after the mutex has already been released by the "normal" program flow, leading to a double unlock and an application crash.

These scenarios illustrate the importance of tools targeted at testing recovery code: such code is hard to exercise in the testing lab without LFI-like tools, and it rarely gets exercised in the field. Yet, whenever it runs, it is expected to run flawlessly.

The second MySQL bug is caused by an uninitialized data structure access after a failed `read`. A related bug, describing a silent failure of MySQL when the `errmsg.sys` configuration file is not found [21], has been acknowledged and fixed. However, if the file exists but reading from it fails for a reason such as a low-level I/O error, MySQL logs the error but nonetheless tries to access an uninitialized data structure and crashes.

When testing MySQL, we started out by using random injection, because MySQL routinely checks function return values, so we wanted to see how robustly it does so. Yet, 1,000 random injection tests targeting different functions caused 35 distinct crashes in MySQL. We analyzed the 35 core dumps and, in this way, found the two bugs presented in Table 1. After writing a specific call stack trigger to reproduce each one of them, we attached a debugger and stepped through the code until the bug manifested; in this way, we were able to connect the injected fault to the bug manifestation.

Custom triggers: To show how triggers can increase testing precision, compared to random injection, we eval-

uate in Table 2 the precision of three injection scenarios. We report the number of times the `close` MySQL bug presented in Table 1 was activated while running 100 times the `merge-big` MySQL test suite component. This also illustrates step-by-step how to build a custom trigger for a particular category of bugs:

1. The first attempt used random injection, with a 10% injection probability in each `close` call. This approach triggered the bug 16 times. Bigger injection probabilities lower the precision, because faults end up being injected in other `close` calls, and execution does not reach the intended target.

2. In our second attempt, we took advantage of "domain knowledge" and used the call stack trigger to inject faults with a 10% probability only in calls issued from the code in the particular file where the bug resides. This scenario triggered the bug 45 times.

3. For the final scenario, we took advantage of a peculiarity of this bug: the `close` call happens after a mutex unlock. Therefore, we injected faults in `close` calls that happen shortly after a mutex unlock, in the hope that the fault will trigger cleanup code that will cause a double unlock. We created a parametrized trigger that allows specifying the maximum distance, in number of lines of code, between the injection site and the last mutex unlock. This trigger, with a distance of 2, reproduced the bug 100% of the time. This excellent precision illustrates our earlier point that triggers need only be "precise enough."

Trigger scenario	Precision
Random (10%)	16%
Random (10%) within bug's file	45%
Close after mutex unlock	100%

Table 2: Precision of three triggers targeting the `close` MySQL bug from Table 1.

Recovery-code coverage: Improving coverage of recovery code is notoriously hard, because exercising such code typically requires errors that appear outside the scope of the developed program and are hard to simulate. Although scenarios that exercise recovery code are rarely encountered in practice, programs that must operate reliably (e.g., servers) should be able to recover gracefully from such faults without corrupting user data or crashing. Since Git and BIND are mature, widely-used applications, we expect them to have recovery code for a large set of possible environment errors.

To assess the coverage improvements that LFI can achieve, we first used gcov and lcov to measure the level of recovery-code coverage obtained by the test suite that ships with each of the applications. We manually identified in the lcov results the recovery code blocks for the functions we target for injection—a tedious job, but necessary for an accurate comparison. We then ran the LFI call analyzer on the two target applications; to be conservative, we trimmed the resulting injection scenarios down to approximately 25 library function calls that are known to fail on occasion (e.g., `fopen`, `read`, `sendto`, `fstat`) and excluded all others. We re-ran the default test suite and measured the new level of coverage. Table 3 shows the results.

	Git	BIND
Additional recovery code covered	~35%	~60%
Additional LOC covered by LFI	429	560
Total coverage without LFI	78.7%	61.2%
Total coverage with LFI	79.6%	61.8%

Table 3: Automated improvement in code coverage.

The fact that, without any human assistance, LFI was able to make the default test suite cover up to an additional 60% of the recovery code in mature applications suggests LFI can offer substantial benefits to testers out-of-the-box. The numbers reported in Table 3 are only a conservative estimate of the improvement in testing, because (a) we did not write any new tests, rather relied on the workload generated by the default test suite; (b) we did not test any of the calls for which there was no recovery code at all, even if there should have been; and (c) we injected faults in only a subset of the library calls made by the applications. Note that the call site analyzer can suggest targets for additional tests, thus helping test developers write tests with less effort.

7.2 Injection Target Identification: Accuracy and Efficiency

There are two ways to identify good injection targets: manually or automatically. We believe the most practical approach is one in which injection targets are first identified automatically, by tools like the LFI call site analyzer, and then developers manually refine the generated injection scenarios. The refinement can be done either based on knowledge of the target system or iteratively, by trying out increasingly focused failure scenarios of interest.

To maximize usefulness of an automated injection target identifier, it must be accurate. The accuracy of injection target identification can be defined as:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

where TP stands for the number of true positives, TN for true negatives, FP for false positives, and FN for false negatives. In the context of injection target identification, these are defined by the following confusion matrix:

LFI says...	Actually checked	Not actually checked
error return is checked	TN	FN
error return is not checked	FP	TP

We used the call site analyzer to identify places in the target system where libc calls are made and the return code is not checked. We then manually inspected the code to cross-check the results (see Table 4). Note that we did not specifically select the ones that are favorable to LFI; we are showing here *all* the calls for which we performed the manual inspection and validation.

System	Function	TP+TN	FN	FP	Accuracy
BIND	malloc	17	0	0	100%
BIND	unlink	6	0	0	100%
BIND	open	5	0	1	83%
BIND	close	39	0	0	100%
Git	malloc	25	0	0	100%
Git	close	127	0	0	100%
Git	readlink	7	0	0	100%
PBFT	fopen	6	0	0	100%

Table 4: LFI's call site analysis accuracy with no human assistance, no documentation, and no source code.

Based on these results, we conclude that LFI's call site analysis is highly accurate for libc calls, even though it is performed directly on x86 binaries; we expect this accuracy to carry over to other libraries beyond libc. It is therefore reasonable to expect that LFI can automatically provide a good set of injection scenarios that developers can then adjust as needed for their tests.

Efficiency: Besides accuracy, running time of the analyzer is also an important factor, because testers are unwilling to wait long for results. For example, it is frequently said that the long running times of model checkers have discouraged their widespread use in testing.

The LFI call site analyzer is fast: in our experiments, analysis time ranged from 1 second to a maximum of 10 seconds for BIND, in cases where there were more than 100 call sites. Analysis time is only influenced by program size (i.e., number of machine instructions) and number of call sites that have to be analyzed.

Developers can process the results of the analyzer fairly quickly. With each call site found, the details regarding file name and line number are provided, if debug symbols are available; this information can guide the developer in inspecting the source code.

7.3   Studying System Behavior

Finding bugs is not the only objective of a tool like LFI—it can also be used to study the behavior of systems under various circumstances. For example, the users of a distributed system may be interested in knowing how it behaves in the face of network failures. We illustrate here the use of LFI for studying the behavior of PBFT's implementation, which is hard to reason about based solely on the design, without experimental evaluation. Our setup consisted of four replicas (i.e., $f = 1$) and one client. We used the *simple_client* and *simple_server* programs shipped with PBFT to generate test workload.

In our first experiment, we used LFI along with a stock distributed trigger (§3.2) to see how PBFT's performance is affected by faults in inter-replica communication. We randomly injected faults in sendto and recvfrom with a variable probability, simulating a degraded (but not malicious) network environment. Using as a baseline PBFT's performance without LFI's interference, we show in Figure 3 how the slowdown varies (averaged over 7 trials) as network conditions worsen. The performance of PBFT deteriorates gradually, reaching a maximum of $4.17\times$ slowdown for a 99% probability of packet loss (i.e., when only one in every 100 network messages make it to the receiver).
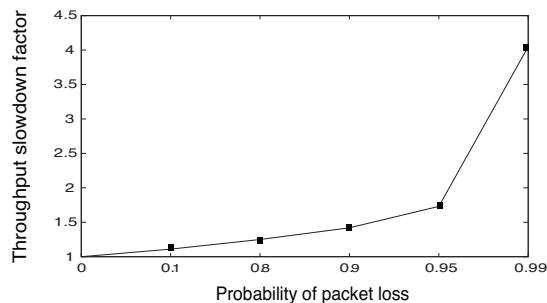

Figure 3: Slowdown in PBFT under progressively worsening network conditions.

While the trend of the curve is not surprising, the exact amount of slowdown experienced at every level of degradation would be difficult to guess without direct measurement.

In a second experiment, we used LFI to simulate a denial-of-service (DoS) attack on PBFT's replicas and we measured again PBFT end-to-end performance, averaged over 7 trials. The baseline for the experiment corresponds to LFI intercepting the calls but letting them all succeed. By injecting consecutive faults in all communication of a specific replica (thus rendering it practically inactive), we obtained an overall performance *improvement* of 12%, possibly due to reduced communication overhead. The third set of measurements corresponds to

an attack in which 500 consecutive faults were injected in replica R_1's communication, then 500 in R_2's, then 500 in R_3's, then again 500 in R_1's, and so on. Such an attack targets the reconfiguration protocol, aiming to confuse it. PBFT's throughput dropped by a factor of $2.2\times$.

The results indicate that the second DoS attack scenario is more effective than the first. While this behavior may not be necessarily surprising, the number of faults and their impact on the overall performance cannot be easily inferred from the design of the system.

As seen in this section, LFI can be used to study the behavior of system implementations under various failure conditions. Since LFI works on binaries, we believe this can be a useful tool for engineers who wish to evaluate, for instance, closed-source third party software, such as databases, load balancers, or application servers.

7.4 The Precision/Performance Tradeoff

It is obvious that LFI triggers can be designed at arbitrary levels of precision, using information in call stacks, program variables, system state, etc. In the context of fault injection, precision denotes the degree to which repeated runs of the target program trigger the same injections. For example, a precise injection would be one that is made only when the system processes a specific request (e.g., a database answering a SQL query), but not when it processes other queries, even if the same library functions are called in the same conditions.

The question we wish to answer is: What is the cost of this precision? If, for instance, the process of injecting library-level faults via LFI slows down the system to the point that its behavior is no longer representative, then the value of testing is decreased (though not eliminated).

To analyze the performance of the trigger mechanism, we measured two commercial-grade servers that are highly performance-sensitive: the Apache 2.2.14 Web server and the MySQL 5.1.44 database server. We computed the induced overhead as a function of number of triggers, frequency of triggering, and type of triggers. We used the Apache benchmark (AB) [1] on Apache and the SysBench [25] Online Transaction Processing benchmark on MySQL.

In order to allow the benchmarks to proceed correctly, we did not actually inject faults, but allowed the triggers to pass the calls through to the real library functions. In this way, we focus the measurement on the triggering mechanism. Our purpose is not to measure how long it takes the applications to recover after encountering a fault, but rather what overhead is introduced by LFI's trigger mechanism.

We constructed injection scenarios with a variable number of triggers and combinations thereof:

- Trigger 1: This trigger targeted `apr_file_read` calls. We used it for targeting calls that have the file descriptor pointing to a socket. The trigger uses the `apr_stat` function on the received file descriptor to check its type.

- Trigger 2: As we wanted to focus testing on Apache's core, and exclude dynamically loaded modules, we also decided to check if the function caller is Apache via the call stack trigger.

- Trigger 3: We further narrowed down our injection target by requiring that the function call happens while processing a request. We used a variation of the stock call stack trigger to require the existence of Apache's `ap_process_request_internal` function in the call stack.

- Trigger 4: We then adapted the stock application state trigger in order to permit injections only when the HTTP *POST* method is used to make the request. To do so, the trigger had to analyze the `request_rec` argument received by Apache's `ap_process_request_internal` function and examine its `method_number` field.

- Trigger 5: We wrote a custom trigger in order to intercept lock and unlock methods in order to target only `apr_file_read` calls made while holding a mutex.

Table 5 summarizes the results, obtained on two different benchmark workloads: static HTML and PHP requests. The former, consisting of 1,000 requests of a static HTML page, fires triggers 32,612 times (about 1.7×10^5 triggerings/second) for the maximum number of five triggers. The second workload is computationally more demanding on Apache, resulting in fewer library calls per unit of time: a total of 45,228 triggerings (about 2.8×10^4 triggerings/second). In both cases, the overheads introduced by trigger evaluation are negligible, suggesting that LFI can be used without affecting the target system's behavior other than through the injected faults.

	Static HTML	PHP
Baseline (no LFI)	0.179 sec	1.562 sec
1 trigger	0.179 sec	1.564 sec
2 triggers	0.179 sec	1.574 sec
3 triggers	0.179 sec	1.577 sec
4 triggers	0.186 sec	1.585 sec
5 triggers	0.188 sec	1.589 sec

Table 5: Running time of the Apache Web server while using LFI with 1-5 triggers. The baseline represents Apache httpd without any interference from LFI.

We also ran the SysBench [25] Online Transaction Processing (OLTP) benchmark on the MySQL RDBMS with LFI applied to GNU libc. We devised injection scenarios with 1-4 triggers on the `fcntl` function:

- Trigger 1: Inject when the `cmd` argument is `F_GETLK`.

- Trigger 2: Inject only when the thread count is bigger than 64 (tests the global variable `thread_count` with the application state trigger).

- Trigger 3: Inject only when the system is shutting down (tests the global variable `shutdown_in_progress` with the application state trigger).

- Trigger 4: Inject when the call is made by the main application module and not other libraries (uses the call stack trigger).

Table 6 shows the results for random triggering and two different workloads: read-only and read-write queries. For the highest number of four triggers, there were ~14K triggerings/second.

	Read-only	**Read/Write**
Baseline (no LFI)	1076 txns/sec	326 txns/sec
1 triggers	1064 txns/sec	319 txns/sec
2 triggers	1060 txns/sec	318 txns/sec
3 triggers	1056 txns/sec	316 txns/sec
4 triggers	1056 txns/sec	316 txns/sec

Table 6: MySQL database server performance while applying LFI with 1-4 triggers (number of transactions per second, as reported by the SysBench OLTP benchmark).

Even with complex combinations of conditions that check various parts of the system state, LFI introduces negligible overhead (consistently less than 5% in our measurements). This offers an advantageous precision/performance tradeoff, meaning that testers can afford to use sophisticated triggers without being concerned that trigger evaluation will bias system behavior.

8 Related Work

Library-level fault injection is an inexpensive testing method first proposed in the context of FIG [6], a tool used to verify the error handling code responsible for GNU libc errors. FIG has several important limitations, in that it only allows injecting faults in GNU libc functions, cannot select particular call sites in which to inject, and requires hardcoding the injected error values.

A refinement was presented by Süßkraut & Fetzer [24] in the form of a system that finds bugs via library-level fault injection and then patches the vulnerable applications to protect against these bugs. Still, this system is limited to GNU libc functions and does not have the means to automatically search for vulnerable call sites, nor to specify complex injection conditions.

This paper continues our previous work on LFI [18], a tool that enables a more general approach to library-level fault injection, by automatically determining meaningful faults to inject and by supporting the interception of arbitrary library functions without the need for source code. To our knowledge, this was the first library-level fault injector practical enough for real-world use. Its main disadvantage, however, proved to be the lack of a mechanism for specifying precise injection conditions, leaving the tester able to only do random or exhaustive exploration of the fault space. The work presented here addresses this shortcoming.

Ideas similar to call site analysis have been previously proposed in [12], where the authors targeted Linux file system implementations at the source-code level and used block-level fault injection to confirm certain categories of bugs. Java exception propagation and handling has been analyzed by Weimer and Necula [27] and Fu et al. [8], using functional specifications and compile-time fault injection, respectively, for discovering bugs. Our approach is complementary, since we target different types of systems, use fault injection at a different level, and operate on binaries.

Other fault injection systems, like G-SWFIT [2], follow another approach to testing: they mutate the target binary code according to statistical bug rules. Different tools operate at even lower levels: FTAPE [26] is designed to inject faults in memory, registers, and disk accesses. NFTAPE [23] can use different low-level fault injectors to test the robustness of systems. Using these systems for testing general-purpose software faces two challenges: it is not clear whether it is reasonable to expect an application to handle such low-level fault (e.g., disk failure), and the large number of layers that separate the low-level injection point from the application level makes pinpointing the location of a possible bug tedious.

The application-library boundary does not suffer from these shortcomings: programs are expected to react properly to error conditions signaled by components with which they interact, and determining the point where a fault transformed into an error is easier, albeit not trivial.

The concept of injection triggers was used by FER-RARI [16] and other tools for OS robustness evaluation [15]. These early trigger mechanisms could only specify a predefined set of conditions, like injecting after the *n*-th function call or after a determined amount of time. The LFI stock triggers and the `Trigger` interface, however, allow testers to achieve a level of precision in recovery-code testing that was previously not available.

9 Conclusion

This paper described a new and improved version of LFI, a library-level fault injection framework that is able to automatically identify errors externalized by shared libraries, identify potentially vulnerable injection targets in application binaries, and produce injection scenarios that exercise such vulnerabilities. The new LFI offers an injection triggering mechanism that allows testers to specify with high precision the conditions under which a fault is to be injected. We presented the stock triggers provided by LFI and the mechanism through which they can be extended to fit practitioners' needs.

LFI was successfully used in testing real systems, and it found 11 new bugs in the BIND name server, the Git version control system, the MySQL database server, and the PBFT replication system. LFI achieved 35%-60% recovery-code coverage entirely on its own, with no human involvement. We have also shown that LFI introduces only negligible runtime overhead during testing.

LFI can be downloaded at `http://lfi.epfl.ch/`.

Acknowledgments

References

[1] Apache Benchmark (AB). `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[2] R. Barbosa, N. Silva, J. Duraes, and H. Madeira. Verification and validation of (real time) COTS products using fault injection techniques. *Intl. Conf. on Commercial-off-the-Shelf (COTS)-Based Software Systems*, 2007.

[3] BIND - [BUG] BIND abort in dst_api.c. `https://lists.isc.org/pipermail/bind-users/2010-January/078493.html`.

[4] BIND - [BUG] BIND crash in statschannel.c. `https://lists.isc.org/pipermail/bind-users/2010-January/078428.html`.

[5] E. Bisolfati, P. D. Marinescu, and G. Candea. Studying application–library interaction and behavior with Lib-Trac. In *Intl. Conf. on Dependable Systems and Networks*, 2010.

[6] P. A. Broadwell, N. Sastry, and J. Traupman. FIG: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems*, 2002.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Symp. on Operating Systems Design and Implementation*, 1999.

[8] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of Java web services for robustness. In *ACM SIGSOFT Intl. Symp. on Software Testing and Analysis*, 2004.

[9] Git - [BUG] crash on make test. `http://marc.info/?l=git&m=125985479417107`.

[10] Git - Git unchecked mallocs. `http://marc.info/?l=git&m=126298802319662`.

[11] Git - Running commands in wrong environment. `http://marc.info/?l=git&m=125986795807036`.

[12] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: error handling is occasionally correct. In *USENIX Conf. on File and Storage Technologies*, 2008.

[13] 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 *Symp. on Operating Systems Design and Implementation*, 2008.

[14] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *USENIX Windows NT Symp.*, 1999.

[15] A. Johansson, N. Suri, and B. Murphy. On the impact of injection triggers for OS robustness evaluation. In *Intl. Symp. on Software Reliability Engineering*, 2007.

[16] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2), 1995.

[17] libdwarf. `http://reality.sgiweb.org/davea/dwarf.html`.

[18] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *Intl. Conf. on Dependable Systems and Networks*, 2009.

[19] MySQL - [BUG] MySQL crash due to double unlock. `http://bugs.mysql.com/bug.php?id=53268`.

[20] MySQL - [BUG] MySQL crash due to error while reading errmsg.sys. `http://bugs.mysql.com/bug.php?id=53393`.

[21] MySQL - [BUG] MySQL crash due to missing errmsg.sys. `http://bugs.mysql.com/bug.php?id=25097`.

[22] M. Prasad and T. Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *USENIX Annual Technical Conf.*, 2003.

[23] D. T. Stott, B. Floering, Z. Kalbarczyk, and R. K. Iyer. A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Intl. Computer Performance and Dependability Symp.*, 2000.

[24] M. Süßkraut and C. Fetzer. Automatically finding and patching bad error handling. In *European Dependable Computing Conference*, 2006.

[25] Sysbench. `http://sysbench.sourceforge.net`.

[26] T. K. Tsai and R. K. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In *Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, 1995.

[27] W. Weimer and G. C. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2), 2008.