Efficient Testing of Recovery Code Using Fault Injection

PAUL D. MARINESCU and GEORGE CANDEA,
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

A critical part of developing a reliable software system is testing its recovery code. This code is traditionally difficult to test in the lab, and, in the field, it rarely gets to run; yet, when it does run, it must execute flawlessly in order to recover the system from failure. In this article, we present a library-level fault injection engine that enables the productive use of fault injection for software testing. We describe automated techniques for reliably identifying errors that applications may encounter when interacting with their environment, for automatically identifying high-value injection targets in program binaries, and for producing efficient injection test scenarios. We present a framework for writing precise triggers that inject desired faults, in the form of error return codes and corresponding side effects, at the boundary between applications and libraries. These techniques are embodied in LFI, a new fault injection engine we are distributing `http://lfi.epfl.ch`. This article includes a report of our initial experience using LFI. Most notably, LFI found 12 serious, previously unreported bugs in the MySQL database server, Git version control system, BIND name server, Pidgin IM client, and PBFT replication system with no developer assistance and no access to source code. LFI also increased recovery-code coverage from virtually zero up to 60% entirely automatically without requiring new tests or human involvement.

1. INTRODUCTION

To build a reliable software system, one must test not only regular code, but also recovery code. Although rarely executed, recovery code must function flawlessly as it is often the last defense before complete system failure. Alas, this code is often unreliable. For example, in the case of the Ariane 5 rocket, a missing exception handler for arithmetic overflows in its control software caused the rocket to self-destruct, leading to a loss of over $370 million [Dowson 1997].

The dominant approach to testing recovery code is to simulate failures that exercise recovery code—this is referred to as fault injection. Alas, injecting meaningful faults is hard. On the one hand, low-level faults (e.g., bit flips) are easy to inject, but are meaningful only when testing applications designed specifically to cope with faulty hardware, which is not the case for most general-purpose software. On the other hand,

higher level faults are more representative, but hard to aim precisely at the recovery code of interest. For example, a tool for simulating lost, reordered, or corrupt network packets [Li et al. 2002] provides a great way to test an operating system's TCP/IP implementation, but exercising the recovery code of general application-level software (e.g., a database or web server) requires that the `read()` call to the C standard library return fewer bytes than expected or return an error code. Simulating a short `read()` via network packet-level injections is difficult because it requires coercing the OS network stack into a specific target behavior.

An alternative for testing against high-level faults is to write special-purpose injection code inside the target system itself. For example, the MySQL database system [MySQL 2010d] has blocks of code that return "fake" I/O errors when activated via a debug directive. This approach is effective, but it requires extensive labor to write the ad-hoc fault simulation stubs, it is not portable from one system to another, and it is difficult to maintain as the system's environment evolves (e.g., upgrades of the libraries or the OS kernel). Furthermore, such ad-hoc testing code can itself contain bugs.

The approach we present in this article focuses on the boundary between applications and their libraries as most error recovery code can be exercised via faults at this interface. Programs interact with their environment through libraries, and most events in the environment, including failures, are exposed to applications through the library APIs. Shared libraries, in particular, are widely used since they encapsulate frequently used functionality (a general-purpose application links to tens or even hundreds of shared libraries [Marinescu et al. 2010]). The reliable functioning of programs is thus tightly coupled to how well they handle error returns from shared libraries. Furthermore, although library APIs are usually well documented, they are complex and often differ from platform to platform in subtle ways (e.g., `errno` values for the same standard C library function often vary between Linux, Solaris, and Mac OS). This invites bugs as such corner cases are easy to miss during development, and improper handling of these error cases can lead to crashes or subtle problems (e.g., if a `read()` call is not retried after receiving an `EINTR` error, the calling application could miss data in its input stream).

Automated testing tools, such as those based on model checking [Killian et al. 2007] or symbolic execution [Cadar et al. 2008], can exercise code in a systematic fashion with no human assistance. The key principle is to exhaustively explore states or exercise paths of a target program, including those dealing with recovery. This approach is thorough, but, in general, it scales poorly because the number of states/paths in a program is typically at least exponential in the program's size. Thus, for systems like MySQL, with over 1 million lines of code (MLOC), it is still challenging to employ automated testing tools. Furthermore, many such tools require access to source code, which may not always be available. In our work, we aim to build a tool suite that requires minimal human assistance, yet can scale to even the largest software systems and does not need access to their source code.

LFI, the tool suite we describe in this article, analyzes statically the binaries of shared libraries, infers the ways in which calls to their exported functions can fail, and produces a corresponding fault profile for each library. It then intercepts the target program's calls to these functions and selectively fails the calls in the manner described in the profile. In order to better focus the injection, LFI performs an initial static analysis of the target program's binary. The results of repeated tests are aggregated by LFI and presented to the human tester. In contrast to the prevalent approaches, LFI decouples the testing of a program from that program's code, thus enabling automation and reuse of fault injection tests across many systems.

Any automated fault injection technique, including LFI, must answer three key questions: what, where and when to inject? Choosing WHAT faults to inject must

be done carefully to ensure they are realistic or else the tester will be overwhelmed by false positives. For example, a "no disk space" error makes sense if the program attempts to write data to disk, but not if the program tries to read from disk or to write to a socket instead of a file. Choosing WHERE in the target program to inject a given fault is equally important and must be done in a way that is sensitive to the target program's logic. For example, injecting a fault on every call to `read()` is counter-productive and achieves low test coverage because the target program ends up unable to make any progress. Finally, WHEN to inject a fault during the execution of the target program matters with respect to focusing the testing efforts and thus improving efficiency. For example, a class of bugs manifest only when `write()` errors are encountered by the flush code during a database system's shutdown [MySQL 2009, 2010c]; testing for these bugs by failing all writes in the flush code, regardless of whether the system is shutting down or not, is inefficient.

To address the "WHAT" question, LFI automatically infers the ways in which library functions can return errors. The automated LFI *library fault profiler* operates directly on shared library binaries and performs two tasks. First, using static analysis of the machine code, it infers the return codes of the functions exported by the library (e.g., it determines that `read()` in libc can return −1 and 0). Second, it infers side effects. This is necessary because, besides error return values, library functions can communicate to callers additional information regarding the error through 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 automatically identified return values and side effects are summarized in the library's fault profile.

To address the "WHERE" question, LFI statically analyzes the target program's machine code and identifies missing or incomplete recovery code. The LFI *callsite analyzer* determines whether error return values are checked by the callers of library functions. For each identified callsite, it uses dataflow analysis to determine for which error code values the return and side effect are checked. An unchecked error code indicates a potential bug to be verified by developers through fault injection.

To address the "WHEN" question, LFI provides a mechanism for specifying with high precision the conditions under which a given call to a given library should experience failure. These conditions form *injection triggers*—predicates on program state (i.e., on global and local variables, callstack, etc.) which must hold for a given fault to be injected. Triggers provide an expressive way for specifying the exact points in a target program's execution when injection events should occur. LFI provides a set of stock triggers for conditions that are often desired during testing, and also offers an interface for writing new triggers and extending existing ones.

To glue together triggers, callsites, and library fault information into complete fault injection scenarios, LFI offers a *fault injection language*. Composing a scenario consists of indicating which fault (WHAT) to inject, in which call to the target library ("where"), and at which point in a program's execution (WHEN). The LFI runtime then executes the injection scenario by running the target program, injecting the faults as desired, and detecting universal failures (crashes, hangs, etc.) as well as application-specific failures encoded in user-provided scripts; it then reports the aggregate results. The fault injection language allows testers to devise sophisticated fault injection scenarios without having to write new test code.

The contributions of this article are illustrated in Figure 1, which also provides a roadmap of the article: a technique for fault-profiling shared library binaries (Section 2), a callsite analysis technique for program binaries (Section 3), a triggering mechanism for fault injections (Section 4), a fault injection language (Section 5), and a general library-call interception technique (Section 6). These contributions
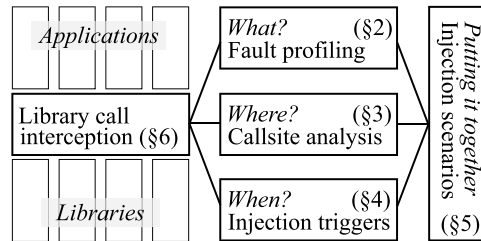
Fig. 1.   The components of the LFI tool suite, along with the sections where they are described.

are embodied in the LFI prototype (Section 7), which we used to find 12 serious, previously unreported bugs in the MySQL database server, Git version control system, BIND name server, Pidgin IM client, and PBFT replication system with no developer assistance and no access to source code. As described in the evaluation of our system (Section 8), LFI increases recovery-code coverage for these systems from virtually zero up to 60% entirely on its own, with no human assistance. In our measurements, LFI incurs negligible runtime overhead. The article ends with a discussion of LFI's limitations (Section 9), a survey of related work (Section 10), and conclusions (Section 11).

2. AUTOMATED FAULT PROFILING OF LIBRARY BINARIES

From the point-of-view of fault injection testing, libraries are in essence a layer through which environment events are filtered, transformed, and propagated to applications. The interface to this layer consists of the functions that libraries export to programs. Constructing a fault profile for a library L therefore consists of finding the ways in which failures can be exposed to programs that call into L.

Doing such profiling *automatically* is crucial because the number of libraries used by general-purpose applications is large, their interfaces are wide, and they change frequently. Furthermore, profiling needs to be redone for every environment in which the target application runs. For example, on BSD systems, the man pages indicate that, whenever `close()` fails, `errno` can be `EBADF` or `EINTR`. On Linux, `errno` can additionally take on the `EIO` value, so programmers porting from BSD to Linux must be aware of this difference and add suitable recovery code for `EIO`. If porting to HP/UX, they must also remember to check for `ENOSPC`, and on Solaris for `ENOLINK`, all of which are return codes present in the corresponding libc libraries. An effective testing tool must help developers deal with these differences.

Another requirement for a practical fault profiler is that it *work directly on binaries* without access to the libraries' source code. First, source code may be unavailable, as is the case for proprietary libraries like most DLLs on Microsoft Windows systems. Second, obtaining source code matching the exact versions of the libraries used is difficult (e.g., the original GNU libc code is slightly different from the version used by RedHat Linux, which in turn differs from the version used by Ubuntu Linux). Third, managing compile and build requirements for many different libraries involves substantial manual work. We believe these three factors would deter practitioners from adopting a tool that required source code.

One possible solution, which is both automatic and independent of source code, is to parse documentation, such as man pages. This approach has two main drawbacks. First, documentation is not always consistent with the library implementation. For example, the `modify_ldt` man page claims three possible return values on Linux/x86 (`EFAULT`, `EINVAL` and `ENOSYS`), yet our LFI library fault profiler found a fourth one (`ENOMEM`), which we confirmed through code inspection. We found similar inconsistencies in `libxml2`, where `htmlParseDocument` is alleged to only return 0 or -1 for success/failure,

```
<profile>
...
<function name="close">
<error-codes retval="-1">
  <side-effect type="TLS" module="libc.so.6" offset="0x12FFF4">-9</side-effect>
  <side-effect type="TLS" module="libc.so.6" offset="0x12FFF4">-5</side-effect>
  <side-effect type="TLS" module="libc.so.6" offset="0x12FFF4">-4</side-effect>
</error-codes>
</function>
...
</profile>
```

Fig. 2. An example fault profile for the `close` function of the standard C library.

yet it can also return 1 in some failure cases. More such inconsistencies have been found recently by Rubio-González and Liblit [2010]. Disparities between documentation and reality are often the very source of program bugs so an effective fault injection tool cannot afford to fall victim to these same disparities. Second, automated analysis of documentation written in natural language is rarely accurate: Descriptions may be incomplete (e.g., "the same errors that occur for link can also occur for linkat" in the `linkat` man page) or ambiguous (e.g., "returns 0 if successful, a positive error code otherwise" appears frequently in the `libxml2` documentation, without actually specifying the value of the error code).

Our approach is different: We employ static analysis to process library binaries and identify the error return values for every exported library function. Moreover, if libraries provide their callers with additional error details through side channels, our library fault profiler identifies them as well. The final output of the profiler is a per-library fault profile. A fault profile maps exported library functions to the sets of error codes that those functions can return, along with possible side channels (e.g., `errno`). Since fault profiles can be used for many other purposes beyond just fault injection testing (e.g., to cross-check API documentation), we adopt an encoding format that is both human-readable and easy to parse.

Figure 2 shows a snippet of the profile generated for the libc `close` function. The profile states that, in case of error, `close` returns -1 and provides additional information via a thread local storage variable (`errno`) at the given offset. This side-effect can be value -9 (corresponding to `EBADF` = bad file descriptor), -5 (for `EIO` = input/output error), or -4 (for `EINTR` = interrupted system call).

The rest of this section presents the algorithm employed by LFI for return-value analysis (Section 2.1), the algorithm for detecting error-reporting side channels (Section 2.2), and an analysis of the algorithms' time and space complexity (Section 2.3).

2.1. Inferring Error Return Values

Given a library L, the goal is to identify for each exported function F_1, F_2, ... F_n the error values that can be returned by each such F_i. These error values may originate inside F_i's code itself or in one of the dependent functions, that is, functions called by F_i. These other functions may be internal to L or external, such as functions in other libraries or system calls exported by the OS kernel. The fault profiler therefore starts by identifying the transitive closure Δ_L of dependencies for L using static analysis of the libraries and system configuration information. The resulting Δ_L contains all the code that LFI needs to analyze.

For each function f contained in the Δ_L code, LFI first determines f's error return values that originate in f. A first question is how to identify potential error values— LFI heuristically assumes error returns are always constants. This approach is

theoretically incomplete (i.e., it can miss error return values that are not constants), but our analysis of the shared libraries on standard Linux systems found that non-constant return values are used only to indicate success conditions (e.g., the number of bytes read). This is not surprising, since error codes are typically defined in header files with `#define` directives or `enum` constructs.

A second question is how to determine, of all *potential* error return values, which ones can *actually* be returned by f. For all application binary interfaces (ABIs) we know of, the return value is placed in a well-defined location (e.g., the Intel ABI specifies that the return value must always be placed in the `eax` register). It is therefore sufficient to find which constants are propagated to this location (either from locally generated constants or by calls to other functions) and remain untouched up to a subsequent return instruction.

To find the true return values among all potential error return values, LFI first constructs for each function f that is part of Δ_L the corresponding control flow graph CFG_f, and then pieces together the CFGs into a global inter-procedural CFG. Then, for each return instruction i (e.g., the `ret` instruction on x86) in each exported function F_i of library L, LFI determines constant values that the return location l (e.g., the `eax` register on x86) could contain at the time of executing return instruction i. We cast the problem of finding the true return values as a multiple-source/single-destination search in the CFG: Find the paths that propagate constants to return location l prior to executing return instruction i. In other words, the algorithm performs "reverse" constant propagation.

Algorithm 1 describes the approach more precisely. As indicated in the input definition, a variable location l can be a memory address, a processor register, or even an immediate value. The algorithm first checks whether the instruction–variable pair has been previously encountered (line 4); this could happen, for instance, when processing loops. If the pair has already been encountered, the empty set is returned (line 5), otherwise the pair is marked as visited (line 6), and the set of values for l is recursively computed as the union of.

(1) l , in the case when l is itself a literal (lines 7–8);
(2) the error returns of the function called by the current instruction i, if i is a call instruction and l is the ABI-specified location for return values (lines 9–11);
(3) the error codes potentially held by variables propagated to l by i (lines 12–14). We say that l' is "propagated" to l by instruction i, and write $l' \xrightarrow{i} l$, if i copies/moves the content of l' to l, or if l' and l refer to the same location and i does not modify it. The *predecessors(i)* function called on line 13 returns all instructions that can execute immediately before i, according to the CFG.

The algorithm employs two additional heuristics (not shown) in order to avoid mistakenly including in the fault profile those constant return codes that represent success. First, it removes 0–return values from all functions for which more than one constant return value was found. If only the 0–return was found, it is likely a null pointer return, so it is preserved in the fault profile. Second, the library fault profiler optionally eliminates functions for which none of the return codes is a failure (as in the case of functions like `isalpha()`) based on an a priori list provided by the tester.

Discussion. Static analysis has limited precision in analyzing the context of a return. Thus, fault profiles may include error codes that can be returned by the corresponding function only under a narrow combination of circumstances. For example, the `read` function in libc can return -1 and set `errno` to `EWOULDBLOCK` only when called with an asynchronous file descriptor, not otherwise. Inferring this type of fine-grained

---

ALGORITHM 1: RevConstPropagate: Determine possible constant values of a given variable at the time of executing a given instruction.

Input: Instruction i
           Variable l (memory address, register, or immediate) in the program
Output: Set R of possible immediate (constant) values that variable l could hold at the time of
            instruction i's execution
static Map:⟨Instruction, Location⟩ → boolean Visited, initially ∅
begin
$\quad$ $R := \emptyset$
$\quad$ **if** *Visited* $[\langle i, l \rangle]$ **then**
$\quad\quad$ **return** ∅
$\quad$ **end**
$\quad$ *Visited* $[\langle i, l \rangle]$:= true
$\quad$ **if** *l is an immediate value* **then**
$\quad\quad$ $R := \{l\}$
$\quad$ **end**
$\quad$ **if** *l is the register for return values, and i is a call to function f* **then**
$\quad\quad$ **foreach** *return instruction i_R in f's body* **do**
$\quad\quad\quad$ $R := R \cup \texttt{RevConstPropagate}(i_R, l)$
$\quad\quad$ **end**
$\quad$ **end**
$\quad$ **foreach** l' *such that* $l' \xrightarrow{i} l$ **do**
$\quad\quad$ **foreach** $i' \in predecessors(i)$ **do**
$\quad\quad\quad$ $R := R \cup \texttt{RevConstPropagate}(i', l')$
$\quad\quad$ **end**
$\quad$ **end**
$\quad$ **return** R
end

---

relationship between arguments and return values is beyond the capabilities of static analysis, but we believe it can be addressed with an approach based on selective symbolic execution [Chipounov et al. 2011].

Since LFI performs inter-procedural constant propagation analysis, the occurrence of indirect calls could pose a challenge. Fortunately, indirect calls are uncommon, even in event-driven and/or object-oriented code as indicated by Prasad and Chiueh [2003] as well as our own experience. Our analysis of real libraries found that only 2.28% of indirect calls (758 out of 33,122) could actually affect the profiler's accuracy in static error code propagation, therefore we do not treat indirect calls specially in our prototype. Should this be a concern, the profiler could be extended to run a points-to analysis and to dynamically resolve indirect calls at runtime and inject the return codes corresponding to the function called.

Certain libraries, such as the C and C++ standard libraries plus any libraries that use them, eventually depend on OS kernel system calls, so many dependent functions reside in the kernel. LFI therefore extends the static analysis to the kernel's binary image as well to identify the error codes that originate in the kernel and may be propagated through the various libraries to the application. For this purpose, LFI treats the kernel binary as a regular library and treats the system calls as a library's exported functions, essentially extending Δ_L.

The fault profiling technique described here is portable: It obtains a library's exported functions and disassembles them using standard tools that ship with most

```
1. call    f8596 <__frame_state_for+0xb96>
2. add     ecx,0x7c91c
3. mov     ecx,DWORD PTR [ecx-0x20]
4. add     ecx,DWORD PTR gs:0x0
5. xor     edx,edx
6. sub     edx,eax
7. mov     DWORD PTR [ecx],edx
8. or      eax,0xffffffff
```

Fig. 3.   PIC code generated by `gcc` for accessing a TLS variable on Linux/x86.

operating systems. To build the CFG, we only need to know which instructions are branches, calls, or returns on the target platform. All other analyses are independent of the platform and ABI, hence it was easy to make the library fault profiler support platforms like Linux/x86, Windows/x86, and Solaris/SPARC (see Section 8.4).

2.2. Inferring Side Channels Used to Communicate Error Codes

Besides error return values, library functions may also communicate to their callers additional error information through *side channels*. LFI automatically identifies the following three categories of side channels:

— thread-local storage (TLS) variables, like the `errno` mechanism used by libc;
— global variables; and
— output parameters (i.e., when the caller of function f passes as argument a pointer to an error information data structure, and f fills it in with error details, if necessary).

Error Codes via TLS Variables. While analyzing the propagation of possible error return codes to a function's ABI-specific return locations, the library fault profiler also scans the enclosing basic block for instructions that write to global or TLS variables. Using the same heuristic described earlier, writes that propagate constant values to these variables are considered to be side channels for error codes.

We illustrate TLS side channel analysis for the most widely used way of building shared libraries, position-independent code (PIC), that is, machine code that can be loaded at any memory address and executed without modifications. In PIC, instructions accessing global or TLS memory addresses always use relative addressing. For instance, on Linux/x86, the function prologue loads the `ebx` or `ecx` register with the instruction pointer, and subsequent code uses the register as a base address for accessing global/TLS variables.

In Figure 3 we show an example from GNU libc, in which a function sets the `errno` TLS variable and places the return value in the `eax` register after receiving an error return from a system call. Line 1 obtains in the `ecx` register the current instruction pointer by calling a helper function. In lines 2–4, the code computes the address of the `errno` variable. Lines 5–6 compute the value to be stored in `errno` as the negative value of `eax` in accordance with the Linux system call convention, and line 7 stores the value into `errno`. Finally, line 8 sets the return value of the function to -1, indicating an error.

To infer the TLS-based side channels, LFI finds all TLS variables used by the analyzed function and applies to each one `RevConstPropagate()` from Algorithm 1, which determines the potential error return codes these TLS locations could hold upon function return. In the example of Figure 3, the library fault profiler first locates line 8, where the constant error return code is created, then detects the side channel by analyzing the containing basic block, and then concludes that exposing this particular error requires LFI to both place -1 in `eax` and set `errno` accordingly.

Table I. Use of Side Channels to Return Error Information from Shared Libraries in Linux

Function	Side channel used for error codes			
return type	TLS variable(s)	Global variable(s)	Output argument(s)	None
void	0%	0%	0%	23.0%
scalar	0.6%	0.4%	3.5%	56.5%
pointer	0.5%	0.5%	3.4%	11.6%

Error Codes via Global Variables. Detecting side channels based on global variables is similar to detecting those based on TLS variables, except the address computation mechanism is slightly different, even if the location of global variables is still computed relative to the instruction pointer. The profiler identifies all memory addresses computed in this way and uses `RevConstPropagate()` to determine the potential error codes they can hold upon function return.

Error Codes via Output Parameters. Error codes can also be returned via structures or objects initially passed to the library function as arguments. Such "output parameters" are found at a well-known location, typically positive offsets from the base stack pointer (when using frame pointers) or other stack/register combinations in general. Positive offsets are outside the current stack frame because the stack "grows" downwards. The library fault profiler looks for writes to addresses obtained via positive offsets and treats them as side channels just as in the case of TLS or global variables. For example, on x86, the library fault profiler looks for constant propagations to locations of the form `[ebp+x]`, where x is a small positive offset, using calls similar to `RevConstPropagate(`*exitInstr*`,[ebp+8])`, where *exitInstr* is the final instruction of the function analyzed.

To get a sense of how frequently these three different mechanisms are used to return error information, we analyzed more than 20,000 functions exported by the libraries with development headers available in Ubuntu 9.04 Linux and obtained the breakdown shown in Table I. Row labels indicate function return type, column labels indicate the method for providing error details, and values in cells indicate the corresponding fraction of all the functions we analyzed. We used the ELSA C/C++ parser [ELSA 2009] to analyze all the library headers, and then combined this information with the automated analyses performed by LFI.

We found that almost 10% of shared library functions use side channels so automated fault injection tools must be able to identify these side channels.

2.3. Complexity Analysis of Fault Profiling

Having shown how to automatically identify the errors exposed by library functions, we now turn our attention to the complexity of performing this analysis.

The time complexity of Algorithm 1 is $O(|I|)$, where $|I|$ is the number of instructions in the analyzed code (i.e., in the transitive closure Δ_L). To understand why, we first observe that the algorithm does a depth-first traversal on a graph where nodes are $\langle instruction, location \rangle$ pairs, and edges result from data transfer instructions. The time complexity of the depth-first search algorithm is $O(|V|+|E|)$, where V is the set of nodes and E the set of edges in the graph. The second observation is that we only need to analyze the strongly connected component generated by the source $\langle i, l \rangle$ so the complexity is dominated by the number of edges $|E|$. Each data transfer instruction corresponds to at most one edge because we do not do a points-to analysis for the operands of the data transfer instructions. (This approximation does not lose precision because, in real systems, return values have a short lifetime and are referred through the same location.) Therefore, the number of edges is bounded from above by the number of data transfer

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

Fig. 4.   Common bug pattern.

instructions, itself bounded from above by the size of the analyzed code. The resulting time complexity is thus $O(|I|)$. This is the best possible asymptotic complexity for the entire algorithm since we necessarily need to read, disassemble, and construct the function CFG, and this requires time linear in the size of the analyzed code.

The space complexity of the algorithm is $O(|I|)$ as well. To reach this result, we start from the underlying search algorithm with space complexity $O(|V|)$. Since the elements of V are generated on-the-fly, only when data transfer instructions affect a new ⟨*instruction, location*⟩ pair, we bound from above the space complexity via a similar argument as in the time complexity analysis, obtaining again a space complexity linear in the size of the code analyzed.

This section described a technique for automatically extracting from shared library binaries information on how those libraries expose failures to their callers. The key idea is to statically analyze the binaries and identify (a) the constant values that can be returned by each exported function, and (b) the side channels through which additional error information can be conveyed to the caller. As we will show in Section 8, these analyses have high accuracy and are fast, taking on the order of seconds to analyze even the largest libraries. Having shown how LFI answers the "WHAT" question of fault injection, we now describe how it answers the "WHERE" question.

3. IDENTIFYING HIGH-VALUE INJECTION TARGETS

A key goal of LFI is to increase the productivity of using fault injection in testing. The LFI callsite analyzer therefore prioritizes the testing of recovery code that appears to be buggy, while deemphasizing code that appears to be fine. This strategy significantly reduces the number of candidate injection points and enables developers to invest testing time where it is most likely to find bugs.

To illustrate the notion of a *high-value injection target*, consider the code snippet in Figure 4, which is a simplified form of a bug we found in several systems, including BIND and Git. Since the return value of `opendir` is not checked, the argument to `readdir` could be a null pointer. 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. The call to `opendir` in this example is a high-value injection target because making `opendir` fail is likely to uncover a bug.

LFI's approach to identifying such injection targets consists of searching the target program binary for callsites where a library function is called, and then using dataflow analysis to determine whether the caller checks for all possible error codes (and corresponding side channels) that the function could return as per the corresponding library fault profile. Callsites where return values or side channels are not properly checked constitute promising places to inject faults.

Algorithm 2 describes a simplified version of the analysis. The inputs are the program binary X, the library function of interest f, and the set E of returnable error codes obtained from the library's fault profile. The output consists of three sets of callsites: The set C_{all} of sites where f's return (or side channel) is checked for all error codes, C_{some} where it is checked for only some of the error codes in E, and C_{none} where it is not checked for any error codes in E.

ALGORITHM 2: Callsite analysis (simplified)

Input: Executable X, function name f, set E of returnable error codes
Output: Set C_{all} of fully checked calls,
          Set C_{some} of partially checked calls,
          Set C_{none} of completely unchecked calls
begin
    $C_{all} := C_{some} := C_{none} := \emptyset$
    $callSites_f :=$ all callsites in X that invoke f
    foreach $site \in callSites_f$ **do**
        $cfg :=$ partial CFG for code that follows $site$
        $\langle Chk_{eq}, Chk_{ineq} \rangle :=$ dataflow analysis on cfg and X
        if $Chk_{eq} \supseteq E \;\vee\; Chk_{ineq} \neq \emptyset$ **then**
          | $C_{all} := C_{all} \cup \{site\}$
        else if $Chk_{eq} \neq \emptyset \;\wedge\; Chk_{eq} \subset E$ **then**
          | $C_{some} := C_{some} \cup \{site\}$
        else
          | $C_{none} := C_{none} \cup \{site\}$
        end
    end
    return $\langle C_{all}, C_{some}, C_{none} \rangle$
end

Lines 2–3 initialize these sets and find the set $callSites_f$ of places in the target binary where f is called. For each such callsite (line 4), we construct a partial control flow graph for the instructions that follow right after the call to f (line 5) in order to determine how the return value and side effects are handled. The number of post-call instructions to analyze is configurable in LFI; we empirically found 100 instructions to be sufficient for building this partial CFG because errors are almost always either checked shortly after a function call or never checked.

We then perform dataflow analysis (line 6) to follow the propagation of the function's return (or side channel) value through the program. We look at all targets to which the value is copied and look at all literals to which this value (or a copy of it) is compared. We iterate through loops as long as the set of copies of the return (or side channel) value increases. In practice, this set usually stabilizes after a few iterations. Since all correct real systems we analyzed do a local check for error conditions before passing the return value to other functions, LFI performs the dataflow analysis only inside function f. However, it could be done inter-procedurally as well.

The dataflow analysis for each site that calls f yields two 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 via equality, then the callsite goes into the set of fully checked calls (lines 7–8). If error codes are checked via inequality, we assume the entire range of error codes is covered, hence the disjunction on line 7. If only some of the error codes in E are checked via equality, then the callsite goes into the set of partially checked calls (lines 9–10). 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 11–12).

The dataflow analysis is described by Algorithm 3, which solves the general problem of finding all constant values to which the return code (or side channel) of a library function is compared to for either equality or inequality.

We employ a modified depth-first traversal of the control flow graph: Starting from a given instruction i and variable l, we search for execution paths that compare i's

---

ALGORITHM 3: FindComparedConsts: dataflow analysis for determining the constant values
to which a variable is compared

Input: Instruction i and variable l (memory address or CPU register)
Output: Sets Chk_{eq}, Chk_{ineq} of constants that l is checked against for equality and inequality,
       respectively
static Map: $\langle$Instruction, Location$\rangle \rightarrow$ boolean Visited, initially $\emptyset$
begin
    $Chk_{eq} := Chk_{ineq} := \emptyset$
    if *Visited* $[\langle i, l \rangle]$ **then**
       | **return** $\emptyset$
    end
    Visited $[\langle i, l \rangle] :=$ true
    if *i is a comparison instr* comp(*l,v*) *and v is an immediate value* **then**
        if *comparison to v is via equality* **then**
           | $Chk_{eq} := Chk_{eq} \cup \{v\}$
        else
           | $Chk_{ineq} := Chk_{ineq} \cup \{v\}$
        end
    end
    foreach l' *such that* $l \xrightarrow{i} l'$ **do**
        foreach $i' \in$ *successors*(i) **do**
           | $\langle Chk_{eq}, Chk_{ineq} \rangle := \langle Chk_{eq}, Chk_{ineq} \rangle \cup$ FindComparedConsts(i', l')
        end
    end
    return $\langle Chk_{eq}, Chk_{ineq} \rangle$
end

---

value (or a copy of it) to an immediate value. For LFI, we solve a specific instance of
this problem, namely, the one in which i is the instruction immediately following the
call to f, and the location l is the function return (or side channel) location.

The algorithm first checks whether the instruction–variable pair $\langle i, l \rangle$ has already
been processed (line 4). If yes, the empty set is returned right away (line 5); otherwise,
the pair is marked as processed (line 6). If instruction i compares location l to an
immediate value v (line 7), then v must be added to either Chk_{eq} or Chk_{ineq}, depending
on whether it is compared via equality (line 9) or inequality (line 11).

Instruction i may propagate location l to one or more location(s) l', that is, i can copy
or move the content of l to l'. In this case, the algorithm adds to Chk_{eq} and Chk_{ineq}
the constants to which location l' is subsequently compared (lines 12–14) since these
constants represent values to which l is being (implicitly) compared.

For brevity, we omit the side-channel analysis, as it is virtually identical to the one
used for return values.

Discussion. Once the analysis is complete, the LFI callsite analyzer automatically
generates two fault injection scenarios, one corresponding to C_{none} and one correspond-
ing to C_{some}. The LFI runtime interceptor (described in Section 6) can then execute
these scenarios and inject the faults at the vulnerable callsites. We expect testers to
start with C_{none} and, after exhausting the corresponding bug vulnerabilities, to move
on to C_{some}. Note that the role of the callsite analyzer is not to check the correctness
of error handling code, but rather to relieve human testers of the burden of choosing
where to inject faults.

Both the time and space complexity of Algorithms 2 and 3 are $O(|I|)$, that is, linear in the size of the code analyzed. The explanation is similar to the one already presented in Section 2.3, due to the similar underlying depth-first search.

It is also possible to adopt a dynamic approach to identifying "vulnerable" code. For example, we developed LibTrac [Bisolfati et al. 2010], a tool that monitors and records all interaction between an application and dynamic libraries. It then employs data mining techniques to identify calls that could be of interest to fault injection, for example, it detects callsites where faults were never encountered during the execution of the default test suite, indicating untested (i.e., not covered) recovery code. An advantage of LFI's static analysis is that it is faster than LibTrac.

This section has shown how LFI automatically finds high-value injection targets in program binaries. The key idea is to use static analysis to find the places where library functions are called but the potential resulting errors are not properly checked, and then to prioritize injecting faults at those callsites. Even though a vulnerable callsite does not necessarily indicate a bug, this technique substantially reduces the space of candidate injection locations; fault injection tests are then used to confirm or refute the hints provided by static analysis. As we show in Section 8, the callsite analysis algorithm presented in this section is highly accurate in practice. Having described how LFI answers the "WHAT" and "WHERE" questions of fault injection, we now describe how it addresses the "WHEN" question.

4. FAULT INJECTION TRIGGERS

LFI provides a mechanism for testers to specify with high precision when exactly in the course of a program's execution a fault should be injected; we refer to this mechanism as an *injection trigger*. Triggers are in essence predicates which, when evaluated by the LFI runtime, indicate whether a given intercepted library function should fail or not; an injection scenario ties triggers to the logic that indicates what fault should be injected where. A trigger can inspect any part of system state: it can look not only at program state but also at environment state. We describe a set of triggers we developed as part of LFI that can be used simply by referencing them in injection scenarios (Section 4.1). LFI also provides an interface (Section 4.2) that allows testers to tailor stock triggers or write their own triggers from scratch (Section 4.3).

4.1. Stock Triggers

We identified seven types of injection triggers that appear to be used most often by LFI users. We wrote generic triggers using the LFI trigger interface that are customizable via parameters specified in a fault injection scenario. The stock triggers can be used for any intercepted library function.

The *callstack-based trigger* allows injecting faults based on whether the current callstack (or part of it) matches a user-specified set of stack frames. By looking at the callstack, the trigger can learn from which program module the call is being made (e.g., from Apache's SSL module) as well as the call path followed by the program to reach the current location. Callsites in stack frames can be specified using offsets within an object file name or program/library binary, file name/line number pairs if available, or combinations thereof. The LFI callsite analyzer (described earlier in Section 3) automatically produces fault injection scenarios that use callstack-based triggers to inject faults in the locations where error checking seems incomplete.

The *coverage-improvement trigger* also looks at the callstack, but it automatically records and stores the callstacks where faults are injected. This enables a strategy of subsequently injecting faults only when new callstacks are encountered with the goal

of exercising new recovery code. We used this trigger in conjunction with the default test suites of mature programs to improve their recovery code coverage from just a few percentage points up to 60%, as will be seen in Section 8.2.

The *program-semantic trigger* injects faults depending on whether a given relationship between program variables holds (e.g., *numConnections==maxConnections*). This trigger allows checking for relationships between both local and global variables, and it can be easily extended with arbitrary operators. In Section 8.5 we show how to specialize this trigger for data structures specific to the Apache Web server.

The *call-count trigger* allows for specifying that an injection should occur exactly on the *n*-th call to a given library function. Besides its obvious use in testing, this trigger can also be used during debugging to replay observed failures in programs that are driven deterministically by interactions with the environment.

The *singleton trigger* injects a fault exactly once. This type of trigger is often combined with other triggers in a conjunction of predicates. For example, combined with a program-semantic trigger, a singleton trigger can ensure that a fault is injected only the first time *numConnections == maxConnections* holds and never again afterwards. Trigger composition is described in more detail in Section 5.2.

The *random trigger* injects a fault with a configurable probability. It can also be augmented with supplementary conditions through composition (Section 5.2).

The *distributed trigger* is used for testing distributed systems. A central controller receives information on intercepted calls (function name, arguments, and callstack) and, based on a global view of the system under test, decides whether a node-local trigger should fire or not. This allows setting up distributed failure scenarios, such as the ones we used for testing the PBFT replication system (Section 8.2).

4.2. Trigger Interface

In addition to stock triggers, LFI also provides an interface that allows testers to write new triggers from scratch. This interface enables triggers to obtain runtime information to decide whether to inject a fault or not, provides means for triggers to pass this decision on to the LFI runtime, and finally allows testers to configure triggers through declarative injection scenarios. This latter point enables trigger reuse, for example, the previously described random trigger can be used to inject faults with a 10% probability just as well as with 80% or any other probability.

Triggers can maintain any amount of state in order to provide the desired semantics. For example, a trigger T may choose to keep track of when in the past it decided to inject a fault or how many times a library function was invoked but T did not choose to induce a failure. A simple case of this is the singleton trigger, which injects a given fault exactly once during the entire program execution.

The LFI runtime informs triggers only about the library function currently being intercepted and its arguments. It is the trigger's responsibility to directly obtain any other information it needs about the environment or program. For example, a trigger can use the GNU libc `backtrace()` function to inspect the callstack and determine whether the intercepted library function was called by a program, by a function in the intercepted library, or by code in some other library. This approach makes triggers able to access any part of the system state, but it has the downside that testers must write the corresponding code. However, LFI mitigates this aspect by offering out-of-the-box most code required for accessing relevant information.

Triggers are independent modules, written as C++ implementations of the `Trigger` interface. The boilerplate code needed for a trigger is minimal: Usually less than 100 lines of code are needed to write a useful trigger. In our initial LFI prototype, prior to the one described in this article, we wrote triggers directly inside the runtime; this

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

Fig. 5.   LFI's C++ interface for defining injection triggers.

made them difficult to extend and required knowledge of LFI internals. With the new `Trigger` interface, shown in Figure 5, this is no longer the case.

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 illustrate by example in Section 5.1.

The `Eval` method is where the main trigger logic resides. It is called every time a library 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` may end up being 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 access source code or documentation, it cannot automatically infer the number of arguments to pass. It is possible though to extend LFI with heuristic techniques for inferring function arguments [Bisolfati et al. 2010; Zhang et al. 2007].

In Figure 6 we illustrate trigger construction with a sketch of how to build a custom trigger. 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 1KB and 4KB, and the calling thread holds a POSIX mutex.

The `Trigger` interface provides access not only to the information that would be available if fault injection was hard-coded in the target system, but also to global execution information, such as the number of calls made so far to a particular library function. Furthermore, unlike hard-coded approaches, LFI decouples the decision of "WHEN" to inject from the target code as well as from the "WHAT" and "WHERE" questions. This makes triggers substantially easier to extend and maintain. In essence, LFI provides a clean separation between tested code and testing code.

4.3. Designing Triggers

When building a trigger using the `Trigger` interface, several design decisions must be made. In this section we provide guidelines for making these decisions.

Since each trigger includes a predicate (e.g., S_ISFIFO(st.st_mode) && size>=1024 && size<=4096 in Figure 6), a decision must be made on how often to evaluate this predicate. One option is to evaluate it every time the trigger's `Eval` function is called as is done in `ReadPipe1K4KwithMutex`. Another option is to track the changes made to the parts of the program and environment state whose values influence the predicate and only re-evaluate the predicate when one of these changes. For example, a trigger T may inspect a portion of a file to make its decision; doing this frequently could be quite expensive. Instead, T can remember the most recent value of the predicate along with the modification time (`mtime`) information; subsequently, T can choose to reinspect the file only when the new `mtime` differs from the cached one, otherwise it returns the remembered predicate value. Clearly, the former approach is simpler and suitable when the trigger is not expected to be invoked often (or when tracking changes to the relevant

```
DECLARE_TRIGGER ( ReadPipe1Kto4KwithMutex )
{
private:
  static __thread int numMutexesHeld;
public:
  ReadPipe1K4KwithMutex() { }       // empty constructor
  bool Eval( const string& libFuncName, ... ) {  // implementation of Triger::Eval
   if( libFuncName == LOCK_FUNC ) {
      ++numMutexesHeld;             // just saw a mutex lock operation
    } else if( libFuncName == UNLOCK_FUNC ) {
      --numMutexesHeld;            // just saw a mutex unlock operation
    } else if( libFuncName == READ_FUNC && numMutexesHeld > 0 ) {
      // we're doing a read while holding at least one mutex
      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);
      // inject only if 'fd' is a pipe and we're reading between 1KB and 4KB
      return( S_ISFIFO(st.st_mode) && size >= 1024 && size <= 4096 );
    }
    return false;
  }
}
```

Fig. 6.   An example custom LFI trigger.

system state adds high overhead); the latter approach is attractive when the trigger depends on state that changes relatively slowly or that can be evaluated incrementally.

In order to minimize runtime overhead, heavyweight checks (e.g., involving network communication or expensive system calls) should be preceded whenever possible by "lightweight" checks that may directly prove the predicate to be false (e.g., local checks). Triggers should resort to heavyweight checks only if the lightweight checks do not decide the value of the trigger condition on their own.

In theory, one should write triggers that achieve perfect precision, that is, 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: The induced runtime overhead can become nonnegligible, and it takes more effort to write a highly precise trigger. In most cases, we favor an approach where triggers are *precise enough*, that is, inject in all targeted situations and perhaps have a couple of false positives. We provide in our evaluation (Section 8.2) an example of a precise-enough trigger that achieves 100% precision on a real system, although in theory it can yield false positives.

Besides lack of precision, another source of false positives is the injection of unrealistic faults. For example, failing an I/O call that was made on a blocking file descriptor and setting errno to EAGAIN is arguably unnecessarily paranoid given that EAGAIN would normally only occur on nonblocking file descriptors. In LFI, such cases can be handled by composing with a trigger that evaluates to true only when the file descriptor supplied to the I/O call is nonblocking (e.g., the trigger can check the file descriptor with fcntl).

LFI's trigger mechanism provides an arbitrarily precise way of specifying "when" to inject faults. One can define triggers in an imperative way (as a C++ class) or declaratively employ the ready-made configurable triggers. Having described how LFI

```
<!-- Make the trigger known to LFI -->
<trigger id="readTrig1K4K" class="ReadPipe1Kto4KwithMutex" />

<!-- Eval trigger for all read() calls; return -1 whenever trigger returns true -->
<function name="read" numargs="3" return="-1" errno="EINVAL">
  <triggerref ref="readTrig1K4K" />
</function>

<!-- Trigger needs to see all locks/unlocks, so intercept all calls and invoke it -->
<function name="pthread_mutex_lock" return="unused" errno="unused">
  <triggerref ref="readTrig1K4K" />
</function>
<function name="pthread_mutex_unlock" return="unused" errno="unused">
  <triggerref ref="readTrig1K4K" />
</function>
```

Fig. 7.   Example scenario for injecting faults in specific calls to `read()`.

addresses the "what", "when", and "where" of fault injection, we now show how these three mechanisms are glued together to form full-fledged tests.

5. PUTTING IT ALL TOGETHER: INJECTION SCENARIOS

LFI fault injection scenarios are written in a scenario description language (Section 5.1) that also enables flexible composition of triggers (Section 5.2) and productive reuse of fault injection scenarios across multiple systems and test suites. To make the testing process efficient, LFI employs several optimizations in the evaluation of triggers aimed primarily at minimizing runtime overhead (Section 5.3).

5.1. Scenario Description Language

Scenarios are constructed using two main constructs: trigger declarations and associations between trigger instances and intercepted library functions. A trigger *declaration* makes a trigger implementation known to LFI and creates a named trigger instance. An *association* links a named trigger instance to an intercepted library function and to a fault (i.e., "what"); the trigger is then evaluated whenever the function is called, and it decides "when" and "where" to inject.

Fault injection scenarios can be written by hand, but we believe practitioners favor using automated tools for generating and modifying these scenarios, such as the callsite analyzer described earlier in Section 3. For scenarios to be both human-readable and machine-readable, we chose for our prototype an XML-based language; the widespread support for parsing and generating XML makes this choice practical. In the remainder of this section, we provide an overview of the language and describe it formally using a BNF grammar.

Consider the earlier example, where we injected faults in `read` only when the corresponding file descriptor was a pipe, the number of bytes requested was between 1KB and 4KB, and a mutex was held by the calling thread. That `ReadPipe1Kto4KwithMutex` trigger can be associated with the relevant library calls as shown in Figure 7.

The `<trigger>` element declares a trigger *instance* identified by the name `readTrig1K4K` and implemented by the C++ class `ReadPipe1Kto4KwithMutex` (written by the tester or chosen from among the LFI stock triggers). The same trigger class can be used for multiple trigger instances in the same scenario and, of course, in other scenarios.

The first `<function>` element creates an association between the `read` library function and the `readTrig1K4K` trigger instance. Whenever `read` is called by the target program, `readTrig1K4K` is asked for a yes/no answer regarding whether to inject a fault or not. To make this decision, `readTrig1K4K` is given by LFI three arguments (`numargs` attribute)

```
scenario        ::= triggerlist associationlist

triggerlist     ::= trigger*
trigger         ::= triggerid class [arguments]
triggerid       ::= string
class           ::= string
arguments       ::= string

associationlist ::= function*
function        ::= name [numargs] return [errno] [triggerreflist]
name            ::= string
numargs         ::= number
return          ::= number
errno           ::= number | symbolic constant
triggerreflist  ::= triggerref*
triggerref      ::= triggerid

string          ::= [0-9A-Za-z_]*
number          ::= [0-9]*
```

Fig. 8.   The BNF grammar for LFI's fault injection scenario description language.

from the original callstack; 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 `readTrig1K4K` returns true, then LFI returns to the caller a return value of -1 (`return` attribute) from `read` and sets the `errno` variable to `EINVAL` (`errno` attribute). The `errno` attribute is an example of LFI's support for commonly used side channels; testers can further extend LFI with their own choice of side channels.

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".

The scenario description language also enables trigger parametrization, that is, allows arguments to be passed to the trigger instance at initialization time. This means, for instance, that one could replace the `ReadPipe1Kto4KwithMutex` class with a new class that takes the upper and lower bound of the number of bytes as arguments instead of hardcoding them to 1KB and 4KB. An example of such a class is the `ReadPipe` trigger class shown in the next section.

The scenario description language has the grammar shown in Figure 8.

Both the trigger list and the association list may be empty; an empty trigger list means faults should be injected every time a function is intercepted, while an empty function list means no library calls are to be intercepted. The `arguments` component provides optional parameters passed to the trigger's `Init` function. `numargs` is an optional element specifying the number of arguments of the original function to pass to the trigger's `Eval` function (by default, none are passed). Finally, `errno` can be expressed both as a symbolic constant, such as `EINTR`, or directly as a number.

Having shown the basic way of tying together triggers and functions, we now describe how multiple triggers can be composed in LFI.

5.2. Trigger Composition

Trigger composition associates multiple trigger instances with a single library function. Associating multiple triggers within one `<function>` declaration specifies a conjunction of the triggers, that is, they all have to evaluate to true in order for a fault to be injected. Triggers can also be composed in a disjunction (i.e., a fault is injected

```
<!-- Declare and initialize a ReadPipe instance -->
<trigger id="readTrig1K4K" 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" numargs="3" return="-1" errno="EINVAL">
  <reftrigger ref="mutexTrig" />
  <reftrigger ref="readTrig1K4K" />
</function>

<!-- Mutex trigger needs to see all 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>
```

Fig. 9.   A fault injection scenario using a parametrized trigger.

whenever one or more triggers in the composition return(s) true) by adding multiple `<function>` elements using the same function `name`, each one associated with a different trigger.

Trigger composition allows broad reuse of triggers and, together with parametrization, encourages writing flexible, general triggers. Consider the earlier pipe `read` example: Instead of using the `ReadPipe1Kto4KwithMutex` 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 at least one mutex.

The scenario in Figure 9 illustrates the composition of two trigger instances, `readTrig1K4K` and `mutexTrig`. The first `<trigger>` declaration illustrates the initialization of the parametrized `ReadPipe` trigger class when the upper and lower bound on the number of bytes are passed to the `Eval` method of the `ReadPipe` trigger.

Besides conjunctions and disjunctions, LFI also supports negation, whereby the result of a trigger is simply "inverted". Using disjunction, conjunction, and negation, LFI users can assemble a wide range of trigger combinations.

Compared to exhaustive fault injection, injection scenarios allow testers to devise concise and precise tests. A fully exhaustive fault injection campaign is infeasible for all but the most trivial programs because the number of possible failure combinations is an exponential function of the number of library function calls. For example, when the Git version control system clones a 1MB repository, it performs on the order of 5,300 library function calls and takes about 0.5 seconds to complete. If we wanted to test only 250 of them (i.e., ~5% of the calls), going through all of the single-function failures on a single machine would take 2 minutes, all two-function failures would take over 4 hours, and all three-function failures would take nearly 15 days, a period unacceptable for most development teams.

More importantly though, exhaustive injection leads to false positives when the program is exposed to failure combinations impossible to encounter in practice. A

tester can use injection scenarios to specify only the cases where a fault is valid and thus avoid spending time on debugging infeasible scenarios.

5.3. Efficient Trigger Evaluation

In order to minimize collateral interference between the testing framework and the system under test, LFI implements several optimizations.

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), evaluation stops immediately after a trigger evaluates to false. This optimization is similar to the short-circuit evaluation used in most modern programming languages. It can also be leveraged to implement specific trigger semantics, for instance, adding the stock singleton trigger to the end of a conjunction ensures that the corresponding fault is injected at most once.

This optimization ensures that LFI evaluates on average a constant number of triggers, regardless of the size of the conjunction. First, consider the simple case of a conjunction consisting of n triggers, each having an independent probability $0 < p \leq 1$ of evaluating to true. The naive approach of evaluating each trigger every time has time complexity $O(n)$. The optimized approach still has worst-case complexity $O(n)$, corresponding to the case when all triggers evaluate to true, but the expected value of the number of trigger evaluations is $O(1)$. Second, this also holds for the general case when each trigger T_i has its own probability $0 < p_i \leq 1$ of being true. In this case, the expected number of evaluated triggers is at most equal to the case when all triggers have the probability $max(p_i)$ of evaluating to true, therefore it is $O(1)$ as well. The same reasoning can be applied to disjunctions as well.

Two other optimizations are at the level of LFI's internal data structures. One ensures that the list of triggers for the currently intercepted function is obtained in $O(1)$ time, that is, it is independent of the size of the fault injection scenario. The other one minimizes runtime overhead during program startup by using lazy initialization: Each trigger is initialized just before it is invoked for the first time.

In summary, we have shown in this section how tests based on fault injection can be specified using fault injection scenarios. The scenarios can be written by hand, gluing together library functions, triggers, and faults with the LFI scenario description language or they can be generated automatically by tools such as the callsite analyzer described in Section 3. In the next section, we describe the final component of LFI's architecture, the mechanism for intercepting library calls.

6. INTERCEPTING LIBRARY CALLS

The interception mechanism present in LFI offers an automated alternative to the prevalent manual form of fault injection. In the manual method, the developer adds a check in the program before each library function call; depending on the outcome of the check, either the library function is called or a specific error code is assigned directly to the return value. LFI moves all this logic outside the tested application by adding a level of indirection between the application and the libraries.

LFI auto-generates shim libraries to intercept library calls. Based on the fault injection scenario, the shim libraries either pass control to the original function or return an error value. The shim libraries have the same API as the original ones but underneath this API they encode the fault injection logic. These libraries are shimmed between the program being tested and the original library(ies); multiple shimmed synthetic libraries can coexist simultaneously. This design means that LFI requires no access to

```
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, and apply them ... */
    return return_code;
  } else {
    /* ... return stack and registers to original values ... */
    __asm__("jmp [original_fn_ptr]");
    /* original function will return to caller */
  }
}
```

Fig. 10.   A typical stub used by LFI in generated shim libraries.

application source code or any application domain knowledge. The tested application is not aware that its calls are being intercepted nor can it differentiate between errors reported by actual library functions vs. injected errors.

LFI coordinates the entire testing process: It interprets the injection scenario, generates the corresponding interception stubs, and combines them with a small runtime to produce a new library. Once the stubs are generated and installed, LFI 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. The script may also test for more sophisticated application-specific properties. This information is collected in a log, which developers subsequently use to diagnose and fix bugs in the program.

The LFI log records each injected error code, 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 pinpoint and fix the bugs that cause test failures. Third-party systems, like R2 [Guo et al. 2008], can be used to replay deterministically all program failures of interest.

7. THE LFI PROTOTYPE

We implemented the call interception mechanism, the fault profiler, callsite analyzer, and trigger system in a prototype that we distribute in source form at `http://lfi.epfl.ch`. So far, LFI has been used by both software companies and research teams to test a variety of general-purpose software systems.

At the heart of LFI is a system-specific library call interception mechanism, similar in spirit to the one employed by FIG [Broadwell et al. 2002]. LFI creates a shim library that exports stub functions with the same name as the ones being intercepted. On UNIX-style platforms (Linux, Solaris, Mac OS, etc.) we take advantage of the dynamic linker and its library preload mechanism [Curry 1994]; on Windows we use Microsoft Detours [Hunt and Brubacher 1999].

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 typical stub is shown in Figure 10.

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 the number of arguments.

The implementation of the static analysis algorithms used by the library fault profiler (Section 2) and callsite analyzer (Section 3) is optimized for performance. After using a third-party tool to disassemble the binary (e.g., `ldd` and `objdump` on Linux and Solaris, `otool` on MacOS, `dumpbin` on Windows), LFI constructs the control flow graph using an algorithm similar to the classic one by Aho et al. [1986]. The algorithm performs two passes through the program. In the first pass, the CFG nodes (i.e., the basic blocks) are determined by identifying the *basic block leaders*. A basic block leader is either the first instruction in the code, a jump target, or an instruction directly following a jump. Each basic block leader starts a basic block containing all instructions up to the next leader. In the same pass, LFI constructs a mapping between program addresses and basic blocks. In the second pass, LFI adds the edges to the CFG. An unconditional jump induces an edge from the basic block containing the jump to the basic block associated with the jump target, while a conditional jump induces an additional edge to the basic block containing the instruction immediately following the jump. The algorithms described in earlier sections then operate on this representation.

While indirect branches can make building the CFG hard, they are rare in shared libraries: We analyzed 9,633 functions in 30 commonly used libraries and found that only 0.13% of branches (104 out of 78,292) are indirect. The LFI prototype currently ignores any CFG incompleteness that results from indirect branches.

LFI must be able to disassemble the libraries in order to analyze them; this may not work if the code is obfuscated. However, Prasad and Chiueh [2003] report that over 99% disassembly accuracy can be achieved on commercial-grade applications. Therefore, LFI assumes the disassembler output is correct and does not attempt to further validate it. Since the disassembler is decoupled from the library fault profiler and callsite analyzer, it is possible to employ the best disassembler available.

Our goals were to have an extensible trigger mechanism and allow developers to simply drop trigger classes into a directory 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()` method. We used a variation of the Registry design 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. LFI instantiates a trigger by searching in this map and using the corresponding factory method.

To maximize ease of use, LFI can directly handle DWARF debug information [Libdwarf 2010] if present in the tested binaries. For example, the callstack trigger allows testers to specify program locations either by their offset in the binary or by file name and line number. Another example is the callsite analyzer, which can provide developers the exact source-code location of a vulnerable callsite.

In the next section we evaluate key properties of this LFI prototype.

8. EVALUATION

LFI's main strength is precision; it allows testers to specify exactly what fault to inject, where to do it, and when to do it. LFI 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 specific program locations. A developer with good knowledge of the target system or application can test it thoroughly using LFI.

When system knowledge is not available, tools like the library fault profiler and callsite analyzer help improve testers' productivity; we focus our evaluation on how LFI can be used productively even without knowledge of the target code. We answer the following questions: Do real systems contain bad error handling code (Section 8.1)?

Table II. Crashes Found Automatically by LFI in 3 MacOS
Applications: Safari, iCal, and Preview

Application	SIGSEGV	SIGABRT	SIGTRAP
Safari	13	2	0
iCal	8	2	2
Preview	12	3	1

Can LFI find bugs in real systems, and how much code coverage improvement can be obtained (Section 8.2)? What is the accuracy and efficiency of the automated identification of vulnerable callsites (Section 8.3)? What is the accuracy of the library fault profiler (Section 8.4)? How much overhead is introduced by LFI when testing real systems (Section 8.5)? LFI found 12 previously unknown bugs in mature, widely used software systems and improved recovery code coverage from virtually none up to 60%. We find that LFI's static analysis techniques have high levels of accuracy, and the overhead introduced by LFI during testing is negligible.

We evaluate LFI on five systems: BIND 9.6.1, MySQL 5.1.44, Git 1.6.5.4, Pidgin 2.5.5 and PBFT 2008-12-09. These represent five different classes of software. BIND is currently one of the most popular Domain Name System (DNS) servers used in the Internet, and it is the de facto standard for most UNIX-based network infrastructures. MySQL is the most widely used open-source database server with 40% market share and over 65,000 downloads every day [MySQL 2010d]. 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. Pidgin [2010] is a popular instant messaging client. PBFT [Castro and Liskov 1999] 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 $3f + 1$ replicas in total.

We use the binary distributions of the systems just mentioned. We resort to source code only for manual confirmation of LFI's results. All experiments reported here are performed on a 4-core 2GHz Intel Xeon server with 4GB of RAM, running Ubuntu 9.04 with Linux kernel 2.6.28, as well as on a 2-core 2.66GHz Intel i7 MacBook Pro with 4GB of RAM, running MacOS X 10.6.8.

8.1. Error Handling Case Study

To see the extent to which poor error handling affects real applications, we used LFI to inject faults into three MacOS applications: the Safari Web browser, the iCal calendar application, and the Preview document viewer.

Since we do not have source code or knowledge of the internal application logic, we use a simple injection methodology: Choose a set of 8 `libc` functions and inject once per-program execution, each time in a different location. We compare locations by means of a custom trigger that analyzes the callstack at the moment of the library call. To exercise the programs automatically, we build several Automator and AppleScript [Cook 2007] scripts: for Safari we load `http://www.yahoo.com` and `http://acid3.acidtest.org`, for iCal we select a particular date to display, refresh all remote calendars, and then modify an event, and for Preview we load a JPEG image and a PDF document.

Table II shows our results. In a total of 504 automated executions, we experience 43 crashes caused by segmentation faults (SIGSEGV column), aborts corresponding to uncaught exceptions or assertion failures (SIGABRT), and debugger traps corresponding to errors that would break into the debugger (SIGTRAP). The number of unique callers invoking failed library functions, roughly equivalent to the number of unique problems in the code, is 12. Due to the absence of source code, we could not analyze the crashes in detail. However, we looked at the disassembly of one particular case

Table III. Bugs Found Automatically by LFI (See Referenced Bug Reports for Details)

System	Bug
BIND	**Crash** if call to `xmlNewTextWriterDoc` fails while a user is retrieving statistics via HTTP [BIND 2010b]
BIND	**Abort** due to incorrectly handled `malloc` return value in method `dst_lib_init` [BIND 2010a]
MySQL	**Abort** after a double mutex unlock, due to a failed `close` [MySQL 2010a]
MySQL	**Crash** due to a failed `read` (error code EIO) while processing `errmsg.sys` [MySQL 2010b]
Git	**Data loss** caused by running an external command with an incomplete environment, due to failed `setenv` [Git 2010b]
Git	**Crash** due to calling `readdir` with a NULL pointer returned by a previously failed `opendir` call [Git 2010a]
Git	**Crash** due to unhandled `malloc` return value on line 567 in `xdiff/xmerge.c` [Git 2010c]
Git	**Crash** due to unhandled `malloc` return value on line 571 in `xdiff/xmerge.c` [Git 2010c]
Git	**Crash** due to unhandled `malloc` return value on line 191 in `xdiff/xpatience.c` [Git 2010c]
Pidgin	**Abort** after calling `g_malloc` with an arbitrary value after a failed `write` call [Pidgin 2009]
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

and found that, after a failed `open` call, Safari does not check the returned value and uses it instead to create a memory-mapped file via `mmap`. This fails, but the program again does not check the return value and dereferences the corresponding null pointer, resulting in an invalid memory access.

We conclude that improper error checking abounds in today's applications, and tools like LFI can help developers find these problems.

8.2. Effectiveness of Testing: Bugs, Precision, 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. In this section, we run the callsite analyzer on target binaries for Linux and directly apply, with no modifications, the injection scenario generated by LFI. Of course, the deeper the knowledge one has of the system tested, the more effectively LFI's injection triggers can be used. However, here we focus solely on what can be done entirely automatically.

8.2.1. Effectiveness. As a first measure of effectiveness, Table III lists 12 previously unknown bugs found by LFI entirely on its own, along with references to the corresponding bug reports we filed. We expect that, in the hands of a knowledgeable human tester, LFI could find substantially more bugs.

We use the last bug in Table III to illustrate the process we follow in these experiments. After running on the PBFT binary, the callsite analyzer generates an injection scenario of which we show a fragment in Figure 11. The scenario targets a "vulnerable" call to the `fopen` library function present at offset `0x8054a69` in the PBFT `simple-server` binary. We then pass this scenario to the LFI runtime, which runs the test with a standard workload. Upon inspecting the report of the test, we find that a replica is crashed due to a segmentation fault; the log indicates the id of the trigger that fired in that particular test case. Based on the trigger and an inspection of the source code, we find that the replica's shutdown code attempted to write a checkpoint to a file without checking that the file was actually open.

The other PBFT bug ("crash when `recvfrom` fails") does not manifest in the debug build but only in the release build, that is, only when compiled with the `-NODEBUG` option.

```
<trigger id="Trig8054a69" class="CallStackTrigger">
   <args>
   <frame>
      <module>bft/bft-simple/simple-server</module>
      <offset>0x8054a69</offset>
   </frame>
   </args>
</trigger>
<function name="fopen" retval="0" errno="EINVAL">
   <reftrigger ref="Trig8054a69" />
</function>
```

Fig. 11.   Fragment of scenario generated by the callsite analyzer for PBFT.

We simulate deteriorated network conditions by injecting faults in `sendto` and `recvfrom` successively in calls made by different replicas: inject fault in a call made by replica R_1, then in a call made by replica R_2, etc. This type of network misbehavior causes a segmentation fault in the view-change phase of PBFT when a replica tries to access a previously committed message. The reason this bug does not manifest in the debug build is that, when the debug flag is on, the PBFT implementation performs some extra checking and halts as soon as a problem is found. The non-debug build skips this check.

The `malloc` bug in BIND and the `close` bug in MySQL represent examples of buggy recovery code. In BIND, the `dst_lib_init` method checks the return value of `malloc` calls and, if any such call fails, it 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" (nonfailure) 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 since it needs to rescue the system from failure.

The second MySQL bug ("crash while processing `errmsg.sys`") is caused by an uninitialized data structure access after a failed `read`. If the `errmsg.sys` MySQL configuration file exists, but reading from it fails for reasons such as low-level I/O errors, MySQL logs the error and skips the initialization of a certain data structure. However, MySQL "forgets" about the initial file problem and subsequently accesses that data structure, which causes MySQL to crash.

When testing MySQL, we started out with the random injection trigger. MySQL is robust and almost always checks function return values so we wanted to see how well it does this. 1,000 random injection tests targeting different functions cause 35 distinct crashes in MySQL. We analyzed the 35 core dumps and found the two bugs presented in Table III. After writing a specific callstack trigger to reproduce each one, we attached a debugger and stepped through the code until the bug manifested again. In this way, we connected the injected fault to the bug manifestation.

To summarize, LFI is able to find new bugs in mature, real systems without requiring source code or knowledge of the tested system's internal logic.

8.2.2. Precision of Custom Triggers. In the context of fault injection, precision denotes the degree to which identical runs of the target program trigger the same injections.

Table IV. Precision of Three Triggers Hunting for the `close`
MySQL Bug Described in Table III

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

For example, consistently injecting a fault in a given call only when the system processes a specific request (e.g., a database answering a specific SQL query), but not for any other server requests, even if the same library function is called, is a precise injection. We define the precision of a trigger relative to a bug or class of bugs, as:

$$Precision = \frac{\#\ test\ runs\ that\ activate\ the\ bug}{total\ \#\ of\ test\ runs}.$$

In other words, a highly precise trigger will decide to inject the fault precisely at the right time and place so as to cause each run of a given test to fail; an imprecise trigger will make poorer choices and cause fewer identical runs of the test to fail.

To see how triggers can increase testing precision compared to random injection, we evaluate the precision of three injection scenarios. Table IV reports the number of times the `close` MySQL bug presented in Table III is activated while running 100 times MySQL's `merge-big` test suite component. This is a test that exercises MySQL paths in the area around the bug we are targeting.

(1) The first scenario uses random injection with a 10% injection probability in each `close` call. This approach triggers the bug 16 times. Perhaps counter-intuitively, larger injection probabilities lower the precision because faults end up being injected in other `close` calls that precede our target so execution takes alternate recovery paths that do not include the target `close` call.

(2) In the second scenario, we leverage a hint about the bug's location and use the callstack trigger to inject faults with a 10% probability only in calls issued from the code in the file where the bug resides. This scenario triggers the bug 45 times.

(3) For the final scenario, we observe that the `close` call happens after a mutex unlock so we inject faults in `close` calls that happen shortly after a mutex unlock in the hope that the fault will trigger cleanup code that then causes the double unlock. We create a parametrized trigger that allows specifying the maximum distance (in number of lines of code) between the injection site and the last mutex unlock. We then configure the trigger with different distances. With a distance of 2, this trigger reproduces the bug 100% of the time. This result exemplifies our earlier argument (Section 4.3) that triggers need not be perfect but only precise enough.

The three preceding steps illustrate a typical sequence of steps for building progressively more precise custom triggers for a given category of bugs.

8.2.3. Recovery-Code Coverage. Improving coverage of recovery code is notoriously difficult because exercising such code requires errors that typically 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) are expected 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 robust recovery code for a large set of possible environment errors.

Table V. Automated Improvement in Lines-of-Code Coverage

	BIND	Git
Additional recovery code covered	60%	35%
Final recovery-code coverage with LFI	63.6%	37.7%
Total code coverage with LFI	61.8%	79.6%
Total code coverage without LFI	61.2%	78.7%

To assess the coverage improvements that LFI can achieve, we first use gcov [Gcov 2010] and lcov [Lcov 2010] to measure the level of recovery-code coverage obtained by the test suite that ships with each of the applications. We manually identify 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 run the LFI callsite analyzer on the two target applications. To be conservative, we trim 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 exclude all others. We rerun the default test suite and measure the new level of coverage (see Table V).

Without any human assistance, LFI makes the default test suites for BIND and Git cover up to an additional 60% of the recovery code. This suggests that even developers of mature software can substantially benefit from LFI out-of-the-box. Quite remarkably, this substantial increase of recovery-code coverage makes only a tiny difference in overall code coverage: an additional 0.6% for BIND and an additional 0.9% for Git. This suggests that total code coverage is a poor way of assessing the quality of test suites for mission-critical servers; for a system like BIND, despite the fact that almost none of the recovery code is tested, overall coverage is essentially the same as when most of the recovery code is covered.

Note that the numbers reported in Table V are a conservative estimate of how much LFI can improve testing because (a) we did not write any new tests, but 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 (i.e., when the necessary recovery code is missing); and (c) we injected faults in only a subset of the library calls made by the applications.

8.3. Callsite Analysis: Accuracy and Efficiency

The callsite analyzer can suggest vulnerable parts of the code to be targeted with new or additional tests, thus helping developers write thorough test suites with less effort. We now evaluate the accuracy and efficiency of this component of LFI.

8.3.1. Accuracy. 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 callsite analyzer, and then developers manually refine the generated injection scenarios either based on knowledge of the target system or iteratively, by trying out increasingly more focused failure scenarios.

To be useful, a tool that automatically identifies injection targets must be accurate so that developers can trust its results. Accuracy is typically defined in terms of the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) as follows.

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

LFI Callsite analyzer says...	Program indeed checks	Program does not actually check
error return is checked	TN	FN
error return is not checked	FP	TP

Fig. 12.   Definition of true/false negatives/positives for the LFI callsite analyzer.

Table VI. LFI's Callsite Analysis Accuracy with No Human
Assistance, No Documentation, and No Source Code

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%

In the context of LFI's callsite analyzer, a true negative occurs when LFI believes the error code *is* checked, and indeed this is so, while a true positive occurs when LFI believes the error code *is not* checked, and indeed this is so. More specifically, we define true/false negatives/positives with the matrix shown in Figure 12.

We use the LFI callsite analyzer to identify places in various target systems where libc calls are made and their return code is not checked. We then manually inspect the code to determine whether LFI's identification is correct or not; the results appear in Table VI. It is worth noting that we show here all the calls for which we performed the manual inspection and validation, that is, we did not specifically select the ones favorable to LFI.

Based on these results, we conclude that LFI's callsite 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 out-of-the-box a good set of injection scenarios that developers can then adjust as needed for their tests.

8.3.2. Efficiency. The callsite analyzer is accurate, but is it fast? 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.

In our experiments, analysis time ranged from 1 second to a maximum of 10 seconds for BIND, and in each of these cases, there were more than 100 callsites. The measurements also confirm our theoretical complexity analysis in Section 3: Callsite analysis time is influenced solely by program size, measured in number of machine instructions.

Developers can process the results of the callsite analyzer fairly quickly. With each callsite deemed to be vulnerable, the details regarding file name and line number are provided (whenever debug symbols are available), and this information guides the developer in inspecting the source code.

In summary, LFI's automated analysis of the binaries to be tested can identify with high accuracy the callsites that are likely to contain corner-case bugs, and its efficiency in doing so makes LFI's callsite analyzer a convenient complement to (or even a replacement for) manual identification of good candidates for fault injection.

LFI Library fault profiler says...	F can indeed return X	F cannot return X
Library function F can return error code X	TP	FP
Library function F cannot return error code X	FN	—

Fig. 13.   Definition of true/false positives/negatives for the LFI library fault profiler.

8.4. Library Fault Profiler: Accuracy and Efficiency

Just like the callsite analyzer, the library fault profiler must also be accurate if it is to be useful in practice because developers will not trust the profiler's results if they turn out to be off the mark. In this context, we define true/false positives/negatives as shown in Figure 13.

A true positive is an error return code that was correctly found, a false positive is a reported error code that cannot actually be returned, and a false negative is a returnable error that was not found. The notion of a true negative is no longer defined because the profiler does not explicitly identify the values that cannot be returned as error codes. We therefore define accuracy of the fault profiler as:

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

The ground truth for deciding what is a false vs. true positive or negative cannot be easily determined because written documentation is not reliable. To obtain precise numbers, we performed labor-intensive manual code inspection. We limit our initial evaluation to a thorough analysis of a small library, libpcre, with 20 exported functions. We find that the library profiler's accuracy is 84%: It has 52 true positives, 0 false positives, and 10 false negatives (due to compiler optimizations that our implementation is not yet tuned for).

As explained in Section 2.1, another factor that can influence accuracy is the number of indirect branches and indirect calls because they pose a challenge to the static construction of the control flow graph. Accuracy can also be influenced by library design: The number of false positives increases as functions maintain more state from one call to another, based on which they decide the appropriate return value.

In order to assess these effects, we broadened our evaluation to 18 additional libraries on 3 different platforms, but this time we used the documentation as the ground truth. We wrote automated parsers for each library's documentation. While this evaluation is inexact (given that documentation is not always correct), it is the only practical method of comparison. In Table VII we show the results of running the library fault profiler on the respective binaries.

In summary, with no access to documentation, source code, or human assistance, the library fault profiler typically achieves 80%–90% accuracy. False negatives result in missed failure scenarios, while false positives result in time wasted by the developer establishing that the injected fault cannot actually occur in reality. Neither of these is particularly problematic as long as their number is small.

In terms of efficiency, the LFI profiler is practical. We measured profiling times ranging from 0.2 seconds for a small library, libdmx, with 18 exported functions and an 8 KB code segment, to 20 seconds for a very large library, libxml2, with 1,612 exported functions and an 897KB code segment. To profile all 1,000+ libraries found on a typical Linux system took us roughly 5 hours. In practice though, such a full system profile is unlikely to be done often, rather testers will profile only the libraries used by the program of interest. When updating a library on the system, which occurs rather

Table VII. Profiler Accuracy with No Human Assistance, No
Documentation, and No Source Code on Linux/x86, Solaris/SPARC,
and Windows/x86. True/False Negatives/Positives are Relative to the
Respective Library Documentation

Library	Platform	Accuracy	TPs	FNs	FPs
libgtkspell	Linux	100%	7	0	0
libhesiod	Linux	100%	10	0	0
libcdt	Linux	100%	15	0	0
libXss	Linux	92%	12	1	0
libnetfilter_q	Linux	92%	24	2	0
libpanel	Linux	91%	21	2	0
libdaemon	Linux	91%	30	3	0
libdns_sd	Linux	89%	50	4	2
libldap	Linux	85%	368	45	21
libgimpthumb	Linux	84%	31	3	3
libxml2	Linux	80%	989	152	102
libao	Linux	80%	12	3	0
libdmx	Linux	76%	26	8	0
libvorbisfile	Linux	75%	133	4	39
libpanel	Solaris	100%	23	0	0
libpctx	Solaris	83%	10	0	2
libxml2	Solaris	81%	1003	138	88
libssl	Windows	87%	164	18	6

infrequently, it takes on the order of minutes to update its fault profile by reanalyzing the updated library and its dependencies.

Profiling time is mainly influenced by code size, that is, by the number of machine instructions. The number of hops in the propagation of return codes to the eax return register (or equivalent) also has an impact, but we found this distance to not exceed 3 instructions in practice—most likely due to compiler optimizations, like constant propagation and constant folding—so its effect is negligible.

8.5. Triggers and The Precision/Performance Trade-Off

LFI triggers can be designed with arbitrary levels of precision, using information in callstacks, program variables, system state, etc. We wish to quantify 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 evaluate the cost of precision in the LFI trigger mechanism, we measure on Linux two commercial-grade servers that are performance-sensitive: the Apache 2.2.14 Web server and the MySQL 5.1.44 database server. We compute the induced overhead as a function of the number of triggers, frequency of triggering, and type of triggers. We use the Apache benchmark (AB) [Apache 2010] on Apache and the SysBench [SysBench 2010] Online Transaction Processing benchmark on MySQL.

We focus measurement on the triggering mechanism by making the triggers pass all calls through to the real library functions without injecting errors, because our purpose is not to measure how long it takes the servers to recover after encountering a fault, but rather to measure the overhead introduced by evaluating LFI's triggers.

For Apache, we construct injection scenarios that combine up to five different triggers that target calls to the Apache Runtime library (APR) as follows.

Table VIII. Apache Web Server Performance on the AB Benchmark while Using LFI
with Up to 5 Triggers. The Baseline Represents Apache httpd Without any
Interference From LFI

Employed Triggers	Static HTML (overhead)	PHP (overhead)
Baseline (no triggers)	0.179 sec	1.562 sec
T_A	0.179 sec (0.0%)	1.564 sec (0.1%)
$T_A + T_B$	0.179 sec (0.0%)	1.574 sec (0.8%)
$T_A + T_B + T_C$	0.179 sec (0.0%)	1.577 sec (1.0%)
$T_A + T_B + T_C + T_D$	0.186 sec (3.9%)	1.585 sec (1.5%)
$T_A + T_B + T_C + T_D + T_E$	0.188 sec (5.0%)	1.589 sec (1.7%)

— Trigger T_A targets `apr_file_read()` calls whose file descriptor argument points to a socket; this trigger uses the `apr_stat` function to check the argument's type.
— Trigger T_B checks that the caller of the intercepted APR function is the Apache core, in order to avoid injecting faults in any calls coming from Apache's dynamically loaded modules; T_B reuses the callstack trigger described in Section 4.1.
— Trigger T_C further narrows T_B's focus down to function calls made from the core while Apache is processing a request. For this, T_C uses the callstack trigger to fire only when Apache's `ap_process_request_internal()` appears on the callstack.
— Trigger T_D further narrows T_C's focus to inject only when the HTTP *POST* method is used to generate the request seen by the server. For this, T_D reuses LFI's default application state trigger to examine the `method_number` field in the `request_rec` argument passed to `ap_process_request_internal()`.
— Trigger T_E is a custom trigger that intercepts and remembers mutex locks and unlocks in order to inject faults only in calls made while holding a mutex.

Table VIII summarizes the measurements obtained with two different benchmark workloads: static HTML and dynamic PHP requests. The former, consisting of 1,000 requests of a static HTML page, fires the triggers 32,612 times (a rate of $\sim 1.7 \times 10^5$ triggerings/second) for the maximum number of 5 triggers. The second workload consists of 1,000 requests of a PHP page which calls the `phpinfo()` function. This is computationally more demanding on Apache and results in fewer library calls per unit of time: a total of 45,228 triggerings (a rate of $\sim 2.8 \times 10^4$ triggerings/second). In both cases, the overheads introduced by trigger evaluation are negligible in testing, maximum 5.0% and 1.7%, respectively, suggesting that LFI does not substantially affect the target's behavior other than through injected faults.

We also run the SysBench benchmark on MySQL with LFI injecting faults in calls to libc. The injection scenarios use up to four triggers targeted at `fcntl()`.

— Trigger T_1 injects an error return code when `fcntl`'s cmd argument is F_GETLK.
— Trigger T_2 injects an error in the call to `fcntl` only when the total thread count exceeds 64. To determine this, the trigger reads the global MySQL variable `thread_count` using the default application state trigger.
— Trigger T_3 injects an error only while the system is shutting down. It checks for this condition by looking at the global MySQL variable `shutdown_in_progress`.
— Trigger T_4 injects an error when the call to `fcntl` is made by the main application module and not by other libraries. For this, it uses the default callstack trigger.

Table IX shows the results of applying combinations of these triggers under two different SysBench workloads: read-only and read/write. For the highest number of 4 triggers, we obtain $\sim 1.4 \times 10^4$ triggerings/second.

Even with complex combinations of conditions that check various parts of the system state, LFI introduces overhead that is consistently less than 5% in our

Table IX. MySQL Database Server Performance While Using LFI with Up to 4 Triggers.
We show Transaction Throughput as Reported by the SysBench OLTP Benchmark

Employed Triggers	Read-Only (t-put drop)	Read/Write (t-put drop)
Baseline (no triggers)	1076 txns/sec	326 txns/sec
T_1	1064 txns/sec (1.1%)	319 txns/sec (2.1%)
$T_1 + T_2$	1060 txns/sec (1.5%)	318 txns/sec (2.5%)
$T_1 + T_2 + T_3$	1056 txns/sec (1.9%)	316 txns/sec (3.1%)
$T_1 + T_2 + T_3 + T_4$	1056 txns/sec (1.9%)	316 txns/sec (3.1%)

measurements. This level of overhead is negligible during testing. This offers an advantageous precision/performance trade-off, meaning that testers can afford to use sophisticated triggers without being concerned that trigger evaluation will bias the behavior of the system under test.

In this section, we showed LFI's effectiveness in finding previously unknown bugs and in improving recovery code coverage from a few percentage points up to 60%. We showed that the callsite analyzer and the library fault profiler are both accurate and efficient and that they enable testers to automate most tasks involved in testing error recovery code. Finally we showed that fault injection triggers are efficient and can be used to specify precisely the what, where, and when of fault injection. We therefore conclude that LFI is a practical tool for testing recovery code, especially given that it requires no domain knowledge or source code access.

9. DISCUSSION

In this section we discuss some of the decisions made during the design, implementation, and evaluation of LFI and look at LFI's limitations and possible alternatives. We first discuss the practical implications of using LFI for testing (Section 9.1), then discuss the changes needed to port our implementation to a different platform (Section 9.2), and finally analyze the implications of exceptions-based error propagation in library-level fault injection (Section 9.3).

9.1. Using LFI in Practice

The main advantage of LFI is that it provides a "surgical" method for injecting faults, thus offering an alternative to random or exhaustive injection. Testing with exhaustive fault injection is generally not practical: Given n candidate library calls, each with an average of k error returns, the number of scenarios in which some of the n calls fail is on the order of k^n. Even for small values of n and k, this quickly yields millions (or more) test possibilities. For a system like MySQL, assuming an optimistic average of 1 minute/test to bring it up, run a test workload, then bring it down, such exhaustive fault injection testing would take years, even decades, to complete. One might argue that one injection per scenario is enough, but in practice such testing is too weak. Conversely, when employing exhaustive fault injection and finding a bug, tracking down the cause of the observed failure may be difficult: Which of the many injections really was the culprit? Callsite analysis and injection triggers focus the testing effort in a way that can save substantial amounts of time.

LFI works by injecting faults into an application and monitoring its behavior for failures. In essence, there exist two classes of failures: those that are universally accepted as bad behavior (e.g., crashes, hangs) and those that are tied to a specification of the system. The current LFI prototype detects only the former; the latter can only be caught by user-provided correctness-checking scripts or as deviations from a specification of correct behavior.

The number of bugs reported in the evaluation is significantly lower than what we expect a tester to find when combining LFI with knowledge of the system under test. First, we used LFI in a fully automatic way so it only looked for "low-level" universal bad behaviors. Second, we tested highly mature software systems, like MySQL, without employing knowledge of their internal logic. Third, the scenarios generated automatically are limited in their complexity. Deeper knowledge of the system under test allows for: (a) devising scenarios based on the semantic assumptions made by the code and (b) creating injection scenarios tightly connected to the workloads used in the tests. Custom triggers allow the tester to further refine the fault injection process, eliminating unnecessary injections and focusing on scenarios that are difficult to otherwise reason about.

9.2. Generalization of the LFI Approach

Injecting faults at the boundary between programs and external functions is a testing technique applicable to virtually all platforms and programming languages. Our LFI prototype addresses a widely used case, namely x86 and x86_64 executables and dynamic libraries. However, the general techniques presented here carry over to other architectures and languages as well. In this section we describe how each LFI component can be implemented for other types of environments.

Fault Profiling works on a disassembled version of the program, which can be obtained most of the time with standard tools. For compiled languages, these usually are disassemblers, while for interpreted languages (e.g. JavaScript, Python) this facility is usually offered by the interpreters themselves (e.g., the JavaScript V8 shell's `--print_code` option [V8 JavaScript Engine 2011]) or by the standard library (e.g., the Python `dis` module [CPython 2011]). Fault profiling can also work with source code, but the simplicity of disassembly or an intermediate representation is often an advantage.

Callsite Analysis is generally straightforward to implement as it has similar requirements to the fault profiler: the availability of disassembly or an intermediate program representation.

Injection Triggers can be reused in other environments because we implemented injection triggers using an XML-based language that is independent of the target architecture or system. XML has broad support on today's platforms.

Function Interception is generally available on every platform. Intercepting library function calls is done via `LD_PRELOAD` on UNIX-style systems like Linux and Solaris, via `DYLD_INSERT_LIBRARIES` on MacOS, and via the Detours library on Windows. For languages where library functions and external dependencies are not invoked via dynamic library calls, other options for function interception are usually provided. Most dynamic languages (e.g., Ruby) offer a direct way of modifying the type system at runtime, making function interception straightforward. Languages that have the aspect-oriented programming paradigm (e.g. Java with AspectJ) offer the ability to intercept function calls via pointcuts and advices.

9.3. Exceptions Instead of Error Return Codes

Besides error return codes, *exception handling* is another technique for dealing with errors. Exceptions are an out-of-band mechanism for bypassing normal program control flow when an error is encountered and directly invoking the corresponding handling code. LFI does not support exception-based error signaling.

However, exceptions are not generally used to pass error information from dynamic libraries to applications due to the lack of a standardized exception handling ABI. In practice, this means that a program and a dynamic library can safely exchange error

information through exceptions only if they are compiled using the same compiler. This is not usually possible because (a) dynamic libraries and applications are provided through different distribution channels, which may use different compilers in the build process, and (b) libraries must be able to interoperate at any time with multiple applications. Therefore, most dynamic library developers choose a C-style interface to maximize interoperability.

One notable exception is the standard C++ library, `libstdc++`, which is a component of the `gcc` toolchain. The functions that it exports use a mix of return values and exceptions to report errors, which works in this particular case because the library is part of the compiler distribution. LFI can be used to inject faults only through return values and side effects. Extending it to the exception handling paradigm involves adapting LFI's fault profiler to detect the possible exceptions a method can throw and inferring the structure of each exception type, possibly using tools such as Howard [Slowinska et al. 2011]. The interception mechanism does not change, but the intercepting code throws the exception previously inferred using the standard C++ `throw` mechanism instead of returning an error code.

Dealing with exceptions in interpreted programming languages is usually easier due to the availability of reflection APIs that allow for creating exception objects on-the-fly. The appropriate values for the exceptions can be obtained through an analysis similar to fault profiling.

10. RELATED WORK

This article described LFI, to our knowledge the first library-level fault injector practical enough for real-world use. LFI builds upon the experience of many prior fault injection tools, and we describe here some of the more representative ones. Since LFI targets general-purpose systems, not real-time or safety-critical ones, we focus on the approaches relevant to this target domain.

Library-level fault injection is an inexpensive testing method that first appeared in FIG [Broadwell et al. 2002], a tool that verifies the error handling code responsible for GNU libc errors. FIG has several important limitations, such as only allowing the injection of faults in GNU libc functions and not allowing the selection of specific callsites in which to inject. Most importantly, though, FIG requires hardcoding the injected error values, which entails significant manual effort. A refinement was presented by Süßkraut and Fetzer [2006] 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. Like FIG, this system is also limited to GNU libc functions and does not have the means to automatically search for vulnerable callsites, nor to specify complex injection conditions via triggers. Ballista [Koopman et al. 1997] approaches the problem from a different perspective: It allows unit-testing of library code by injecting faults in library function arguments.

There exist fault injection testing tools that target the library or function call interface for specific classes of applications: TestApi [TestApi 2010] for .NET applications, the Hadoop Fault Injection framework [Hadoop FI 2010] for Hadoop's distributed file system, Holodeck [2010] for Windows applications, etc. These tools generally do not assist testers in deciding WHAT, WHERE and WHEN to inject; they either perform "blind" fault injection or require testers to have extensive knowledge of the target application's internal structure. One of LFI's key aims is to provide an entire tool chain that assists testers in all the steps required for fault injection, thus improving their productivity.

Many tools inject faults at layers below the library layer [Barton et al. 1990; Kanawati et al. 1995; Ng and Chen 1999; Stott et al. 2000; Tsai and Iyer 1995].

These tools have been used to test or validate the robustness of systems and protocols [Dawson et al. 1997], and there exist well developed methodologies for this [Arlat et al. 1990]. When employing low-level injection tools for general-purpose software, one typically faces two challenges: It may not be reasonable to expect an application to handle such low-level faults (e.g., BIND cannot be expected to recover from arbitrary memory chip failures), 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 in complex systems. This leads both to false positives (i.e., observed failures that are not caused by application bugs, rather by the developer's decision to not take into account hardware failures) and to difficulty in pinpointing the cause of observed failures. The application/library boundary exploited by LFI does not have 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.

Other fault injection systems [Chillarege and Bowen 1989; Barbosa et al. 2007] follow another approach to testing: They mutate the target binary code according to statistical bug models or rely on human-provided pre- and postconditions that are then purposely violated by dynamically modifying the program state [Bieman et al. 1996], in an attempt to simulate real failures. In a similar vein, tools and techniques have been proposed [Prabhakaran et al. 2005; Bairavasundaram et al. 2008; Gunawi et al. 2010] that employ deep system knowledge in order to trim down the space of faults to be injected; they aim to avoid injecting faults that are known to be irrelevant. LFI's callsite analysis is an example of doing such trimming of the fault space without any system knowledge but rather using automated program analysis. Nevertheless, we expect that techniques that leverage system knowledge can significantly strengthen LFI, for instance, by eliminating false positives caused by the fact that LFI does not track library state across calls.

Ideas similar to callsite analysis have also been proposed by Gunawi et al. [2008]. The authors targeted Linux file system implementations at the source-code level and used block-level fault injection to confirm several categories of bugs. Tools for analyzing the propagation of Java exceptions have also been described by Candea et al. [2003], Fu et al. [2004], as well as Weimer and Necula [2008]. These tools use functional specifications that are known a priori. The tools then use either runtime or compile-time mechanisms to cause the desired exceptions to manifest; the propagation of exceptions uncovers bugs during testing. Our approach is complementary: We target different types of systems and use fault injection at a different level, while not assuming any a priori knowledge of the target software.

Several studies found that documentation, although relied on by programmers, is rarely kept up-to-date as code changes [Lethbridge et al. 2003; Singer 1998]. This has led to proposals for extracting possible function error return values from the source code, such as Rubio-González and Liblit [2010]. This approach focuses on file systems and discovers mismatches between the documentation and the actual code, proving once more that documentation can be inaccurate. In contrast to approaches that rely on source code, LFI's library fault profiler directly analyzes the libraries' binaries, thus it is a good fit for closed-source libraries, too.

The concept of injection triggers was used by FERRARI [Kanawati et al. 1995] and other tools for OS robustness evaluation [Johansson et al. 2007]. These early trigger mechanisms were relatively coarse as they could only specify injection criteria such as injecting after the n-th function call or after a determined amount of time. LFI offers substantially higher flexibility in targeting injections, and LFI's stock triggers and the `Trigger` interface allow testers to achieve a level of precision in recovery-code testing that was previously not available.

11. CONCLUSION

This article described LFI, a reusable and scalable library-level fault injection framework. It consists of a complete tool chain that helps answer the three key questions that arise when employing fault injection: what, where, and when to inject. The LFI fault profiler employs static analysis of library binaries to infer the possible error codes and side effects that result from calling the exported library functions. The LFI callsite analyzer identifies in a program's binary the vulnerable spots where that program fails to check all manifestations of errors in the called libraries. Finally, LFI injection triggers allow testers to specify with high precision where and when to inject a fault. LFI comes with a set of default stock triggers as well as a mechanism for extending these triggers to fit practitioners' needs.

We used LFI successfully in testing real systems. LFI found entirely on its own 12 new bugs in the BIND name server, the MySQL database server, the Git version control system, the Pidgin IM client, and the PBFT replication system. LFI also achieved 35%-60% recovery-code coverage, with no human involvement. We have shown that LFI introduces only negligible runtime overhead during testing.

LFI is freely available and can be downloaded from `http://lfi.epfl.ch`.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

APACHE. 2010. Apache Benchmark (AB). `http://httpd.apache.org/docs/2.0/programs/ab.html`.

ARLAT, J., AGUERA, M., AMAT, L., CROUZET, Y., FABRE, J.-C., LAPRIE, J.-C., MARTINS, E., AND POWELL, D. 1990. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Engin. 16,* 2.

BAIRAVASUNDARAM, L. N., RUNGTA, M., AGRAWAL, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SWIFT, M. M. 2008. Analyzing the effects of disk-pointer corruption. In *Proceedings of the International Conference on Dependable Systems and Networks*.

BARBOSA, R., SILVA, N., DURAES, J., AND MADEIRA, H. 2007. Verification and validation of (real time) COTS products using fault injection techniques. In *Proceedings of the International Conference on Commercial-off-the-Shelf-Based Software Systems*.

BARTON, J., CZECK, E., SEGALL, Z., AND SIEWIOREK, D. 1990. Fault injection experiments using FIAT. *IEEE Trans. Comput. 39,* 4.

BIEMAN, J. M., DREILINGER, D., AND LIN, L. 1996. Using fault injection to increase software test coverage. In *Proceedings of the International Symposium on Software Reliability Engineering*.

BIND. 2010a. BIND aborts in dst_api.c.
`https://lists.isc.org/pipermail/bind-users/2010-January/078493.html`.

BIND. 2010b. BIND crashes in statschannel.c.
`https://lists.isc.org/pipermail/bind-users/2010-January/078428.html`.

BISOLFATI, E., MARINESCU, P. D., AND CANDEA, G. 2010. Studying application–library interaction and behavior with LibTrac. In *Proceedings of the International Conference on Dependable Systems and Networks*.

BROADWELL, P. A., SASTRY, N., AND TRAUPMAN, J. 2002. FIG: A prototype tool for online verification of recovery mechanisms. In *Proceedings of the Workshop on Self-Healing, Adaptive and Self-Managed Systems*.

CADAR, C., DUNBAR, D., AND ENGLER, D. R. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposuim on Operating System Design and Implem.*

CANDEA, G., DELGADO, M., CHEN, M., AND FOX, A. 2003. Automatic failure-path inference: A generic introspection technique for software systems. In *Proceedings of the Workshop on Internet Applications*.

CASTRO, M. AND LISKOV, B. 1999. Practical Byzantine fault tolerance. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

CHILLAREGE, R. AND BOWEN, N. S. 1989. Understanding large system failures - a fault injection experiment". In *Intl. Symp. on Fault-Tolerant Computing*.

CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

COOK, W. R. 2007. Applescript. In *Proceedings of the Conference on History of Programming Languages*.

CPYTHON. 2011. `http://www.python.org/`.

CURRY, T. W. 1994. Profiling and tracing dynamic library usage via interposition. In *Proceedings of the USENIX Summer Technical Conference*.

DAWSON, S., JAHANIAN, F., AND MITTON, T. 1997. Experiments on six commercial TCP implementations using a software fault injection tool. *Softw. Pract. Exper. 27*.

DOWSON, M. 1997. The Ariane 5 software failure. *ACM SIGSOFT Softw. Engin. Notes 22,* 2.

ELSA. 2009. `http://www.eecs.berkeley.edu/~smcpeak/elkhound/sources/elsa/`. (Accessed on 3/09).

FU, C., RYDER, B. G., MILANOVA, A., AND WONNACOTT, D. 2004. Testing of Java web services for robustness. In *Proceedings of the International Symposium on Software Testing and Analysis*.

GCOV 2010. GCC coverage testing tool. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

GIT. 2010a. Git crashes on make test. `http://marc.info/?l=git&m=125985479417107`.

GIT. 2010b. Git fails when running commands in wrong environment. `http://marc.info/?l=git&m=125986795807036`.

GIT. 2010c. Git unchecked malloc's. `http://marc.info/?l=git&m=126298802319662`.

GUNAWI, H. S., RUBIO-GONZÁLEZ, C., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LIBLIT, B. 2008. EIO: Error handling is occasionally correct. In *Proceedings of the USENIX Conference on File and Storage Technologies*.

GUNAWI, H. S., DO, T., JOSHI, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SEN, K. 2010. Towards automatically checking thousands of failures with micro-specifications. Tech. rep. UCB/EECS-2010-98, University of California.

GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. 2008. R2: An application-level kernel for record and replay. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

HADOOP FI. 2010. Hadoop Fault Injection framework. `http://hadoop.apache.org/hdfs/docs/r0.21.0/faultinject_framework.html`.

HOLODECK. 2010. Win32 fuzz testing and fault injection. `http://www.securityinnovation.com/holodeck/`.

HUNT, G. AND BRUBACHER, D. 1999. Detours: Binary Interception of Win32 Functions. In *Proceedings of the USENIX Windows NT Symposium*.

JOHANSSON, A., SURI, N., AND MURPHY, B. 2007. On the impact of injection triggers for OS robustness evaluation. In *Proceedings of the International Symposium on Software Reliability Engineering*.

KANAWATI, G. A., KANAWATI, N. A., AND ABRAHAM, J. A. 1995. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Comput. 44,* 2.

KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. 2007. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the Symposium on Networked Systems Design and Implementation*.

KOOPMAN, P., SUNG, J., DINGMAN, C., SIEWIOREK, D., AND MARZ, T. 1997. Comparing operating systems using robustness benchmarks. In *Proceedings of the International Symposium on Software Reliability Engineering*.

LCOV. 2010. LTP gcov extension. `http://ltp.sourceforge.net/coverage/lcov.php`.

LETHBRIDGE, T. C., SINGER, J., AND FORWARD, A. 2003. How software engineers use documentation: The state of the practice. *IEEE Softw. 20,* 6.

LI, X., MARTIN, R., NAGARAJA, K., NGUYEN, T. D., AND ZHANG, B. 2002. Mendosus: A san-based fault-injection test-bed for the construction of highly available network services. In *Proceedings of the Workshop on Novel Uses of System Area Networks*.

LIBDWARF. 2010. Libdwarf. `http://reality.sgiweb.org/davea/dwarf.html`.

MARINESCU, P. D., BANABIC, R., AND CANDEA, G. 2010. An extensible technique for high-precision testing of recovery code. In *Proceedings of the USENIX Annual Technical Conference.*

MYSQL. 2009. MySQL crashes due to bus error during shutdown. `http://bugs.mysql.com/bug.php?id=42109`.

MYSQL. 2010a. MySQL crashes due to double unlock. `http://bugs.mysql.com/bug.php?id=53268`.

MYSQL. 2010b. MySQL crashes due to error while reading errmsg.sys. `http://bugs.mysql.com/bug.php?id=53393`.

MYSQL. 2010c. MySQL InnoDB crashes during shutdown. `http://bugs.mysql.com/bug.php?id=52546`.

MYSQL. 2010d. `http://www.mysql.com/`.

NG, W. T. AND CHEN, P. M. 1999. The systematic improvement of fault tolerance in the Rio file cache. In *Proceedings of the International Symposium on Fault-Tolerant Computing*.

PIDGIN. 2009. Pidgin SIGABRTs on memory alloc. `http://developer.pidgin.im/ticket/8672`.

PIDGIN. 2010. Pidgin. http://www.pidgin.im.

PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2005. IRON file systems. In *Proceedings of the Symposium on Operating Systems Principles*.

PRASAD, M. AND CHIUEH, T. 2003. A binary rewriting defense against stack-based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference.*

RUBIO-GONZÁLEZ, C. AND LIBLIT, B. 2010. Expect the unexpected: Error code mismatches between documentation and the real world. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*.

SINGER, J. 1998. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance*.

SLOWINSKA, A., STANCESCU, T., AND BOS, H. 2011. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium.*

STOTT, D. T., FLOERING, B., KALBARCZYK, Z., AND IYER, R. K. 2000. A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings of the International Computer Performance and Dependability Symposium.*

SÜßKRAUT, M. AND FETZER, C. 2006. Automatically finding and patching bad error handling. In *Proceedings of the European Dependable Computing Conference*.

SYSBENCH. 2010. `http://sysbench.sourceforge.net`.

TESTAPI. 2010. Library of test and utility APIs. `http://testapi.codeplex.com/`.

TSAI, T. K. AND IYER, R. K. 1995. Measuring fault tolerance with the FTAPE fault injection tool. In *Proceedings of the International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*.

V8 JAVASCRIPT ENGINE. 2011. `http://code.google.com/p/v8/`.

WEIMER, W. AND NECULA, G. C. 2008. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst. 30,* 2.

ZHANG, J., ZHAO, R., AND PANG, J. 2007. Parameter and return-value analysis of binary executables. In *Proceedings of the Annual Intrenational Computer Software and Applications Conference*.