LFI: A Practical and General Library-Level Fault Injector

Paul D. Marinescu and George Candea
School of Computer and Communication Sciences
EPFL
Lausanne, Switzerland

Abstract

Fault injection, a critical aspect of testing robust systems, is often overlooked in the development of general-purpose software. We believe this is due to the absence of easy-to-use tools and to the extensive manual labor required to perform fault injection tests. This paper introduces LFI (Library Fault Injector), a tool that automates the preparation of fault scenarios and their injection at the boundary between shared libraries and applications. LFI extends prior work by automatically profiling fault behaviors of libraries via static analysis of their binaries, thus reducing the dependence on human labor and perfect documentation. We present techniques for automatically generating injection scenarios and we describe a simple language for expressing such scenarios. LFI does not require access to libraries' source code and works for Linux, Windows, and Solaris on x86 and SPARC platforms.

1. Introduction

General-purpose applications rely heavily on shared libraries. For example, we found that the MySQL database server directly links to 13 shared libraries, the Apache Web server can link to more than 30 shared libraries depending on compile options and Adobe Photoshop directly links to 36 shared libraries. If we count recursively the shared libraries used by libraries themselves, the numbers are as high as 138 in the case of Adobe Photoshop.

These applications make important assumptions about how the underlying libraries work, and any guarantees they try to provide to users depend heavily on the correctness of such assumptions. For software that is expected to be highly dependable (database servers, Web servers, email clients, etc.) testing must verify that the ways in which applications use these libraries is consistent with the actual library behavior. In particular, it is essential to verify that the applications correctly handle faults at or below the library layer that manifest as errors returned by the library functions.

Unfortunately, corner cases are easy to miss and can lead to crashes or correctness violations, such as when the result of a memory allocation is not checked, or when a `read()` call is not retried after getting an `EINTR` return code. These bugs are hard to find through input testing, because they are triggered by low-probability events that are typically input-independent and occur below the library layer. To test program robustness, we wish to simulate such error events at the program/library interface and then observe the program's reaction. Ideally, the simulation should be minimally invasive and should not require access to proprietary portions (e.g., source code) of the program or library.

The challenge, though, is that regular systems have an overwhelming number of libraries: a typical Linux system has ~1000 libraries, Windows XP ~1400, and Windows Vista ~1650. To our knowledge, current library fault injectors require considerable amounts of manual work and are restricted to the C standard library (libc), thus not scaling to test all libraries used by programs. Library fault injection must therefore be generalized and automated to the utmost, or else the scope of testing will have to stay narrow.

Not only is it necessary to automate the injection of faults, but also the *inference* of the libraries' fault profiles. Libraries can change frequently; e.g., GNU libc, perhaps the most widely used shared library, has already seen two releases in the first three months of this year [8]. By using shared libraries, applications accept that these libraries may change underneath them; yet, can they suitably cope? Frequent changes can introduce unexpected new behavior, much of which may not even be documented. While many libraries aim for backward compatibility, even GNU libc has not always guaranteed compatibility.

Relying on documentation to decide how a library may expose faults is risky: even if the documentation exists and is correct for one library version, it can get out of sync with the next one. As we show in §3.1, library documentation can be incomplete and miss some of the error return codes. We must therefore extract information on the potential errors directly from the libraries; since source code is often not available, the library binaries themselves must be analyzed. Finally, since libraries can have hundreds of functions in their API, we must automatically generate test scenarios and allow developers to tweak them, instead of requiring them to write tests from scratch.

In this paper we introduce LFI, a tool for automated library fault injection. We wish to make fault injection easy to adopt in the development of general-purpose software, where programmers must be very agile and are subject to constraints different from those encountered in building safety-critical systems. LFI consists of two parts: a profiler and a controller. The *profiler* uses static analysis of libraries' binaries to extract their fault profiles and to determine the side channels used to communicate failure information (such as `errno` in libc). The profiler also generates fault injection scenarios. The *controller* uses profile information to synthesize an interceptor library that can then drive automated injection of fault scenarios.

In the rest of this paper we provide an overview of LFI (§2), describe the LFI profiler (§3), fault scenario generation (§4), and the LFI controller (§5). We then evaluate LFI (§6), survey related work (§7), and conclude (§8).

2. System Overview

The goal of the LFI fault injector is to give testers a fast, easy and comprehensive method to test program robustness in the face of failures that are exposed at the interface between shared libraries and the programs under test. We envision LFI being used not only by testers and researchers evaluating their software prototypes, but also by customers who want to validate closed-source products, or in benchmarks that compare in a systematic way the fault-tolerance of different applications. LFI can also be used as an exploration tool, to understand the behavior of third-party code. LFI can be downloaded from `http://lfi.epfl.ch/`.

Using LFI consists of two steps: (a) profile the target application's shared libraries to determine a set of meaningful faults to inject, and (b) conduct fault injection experiments using various fault scenarios. This is reflected in LFI's architecture (Figure 1).

Testers point LFI at a target application and the profiler automatically finds which shared libraries the application links to and then profiles them. For each library, it determines the exported functions and, for each exported function, the possible error return values—we refer to this information as the library's *fault profile*. LFI does not require symbols and works on both stripped and unstripped libraries; of course, for a library to be useful, libraries must provide symbols for their exported function signatures.

Since profiles are obtained automatically, testers do not need to be familiar with the internals of the libraries. However, if they do have such knowledge or additional domain-specific information, they can alter the generated profiles to obtain faster, more accurate results (e.g., by removing functions or faults that are not of interest).

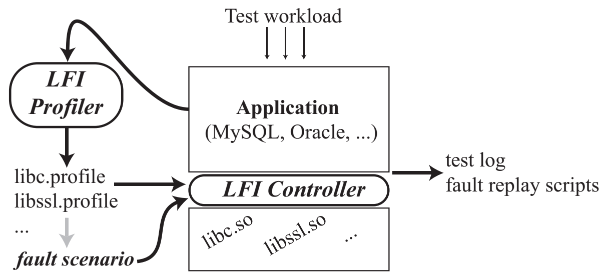


Figure 1. Architecture of the LFI fault injector.

The LFI controller receives these fault profiles and combines them with a fault scenario specification to drive the fault injection. The controller is a transparent shim interposed between the application and the libraries; it intercepts the calls to libraries and injects the desired error codes. In order to be useful "out of the box", LFI automatically generates a set of simple fault scenarios—exhaustive injection and random injection—so, in many cases, testers need not do any manual work. The scenarios can, however, be freely modified after the automatic generation. We show in §6 how random fault injection found a previously-unknown bug in Pidgin, a popular instant messenger client.

The output of LFI experiments is a test report and generated scripts that can replay the injections, enabling developers to debug and test in more detail the scenarios of interest. The results in the report can pinpoint bugs or weak spots in the target software that may be good targets for further examination. The replay scripts can then be incorporated in regression test suites of the target system.

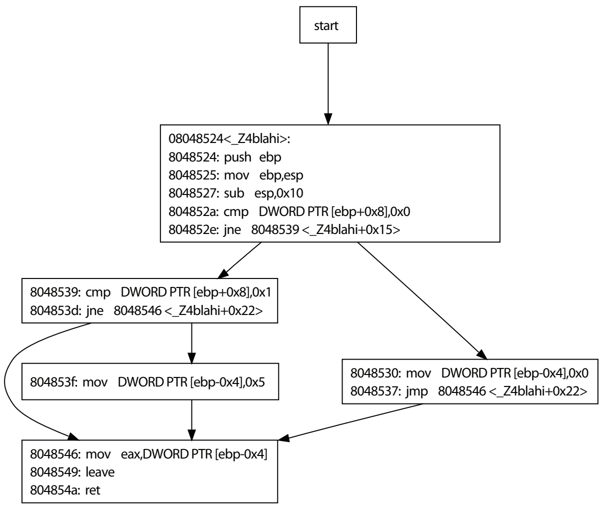
3. LFI Profiler

The interface of a library consists of a set of functions "exported" to programs that use the library. The LFI profiler statically analyzes the library to identify the error return values for every exported function (§3.1). Some libraries provide additional details about error conditions through various side effects; LFI identifies these side channels as well (§3.2). The LFI profiler then outputs a fault profile (§3.3).

3.1. Return Code Analysis

We designed LFI to work directly on the libraries' binaries, because requiring access to source code would hamper the practicality of LFI. First, source code may be unavailable, as is the case for most of the DLLs on Microsoft Windows systems. Second, obtaining source code matching the exact versions of the libraries being used may be 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, handling large source code bases for all the required libraries, with each one having its own set of compile and build requirements, would involve substantial manual work. We believe this would deter practitioners from adopting LFI.

The LFI profiler disassembles the library and identifies all exported functions, along with the dependent functions, i.e., other internal or exported functions that are invoked by the exported functions. Dependencies are determined recursively, both within the same library and other libraries called by the current one. It then constructs for each function a control flow graph (CFG), like the one shown in Figure 2. LFI uses platform-specific tools, such as `ldd` and `objdump` on Linux and Solaris, and `dumpbin` on Windows.



```
08048524<_Z4blahi>:
8048524: push   ebp
8048525: mov    ebp,esp
8048527: sub    esp,0x10
804852a: cmp    DWORD PTR [ebp+0x8],0x0
804852e: jne    8048539 <_Z4blahi+0x15>

8048539: cmp    DWORD PTR [ebp+0x8],0x1
804853d: jne    8048546 <_Z4blahi+0x22>

804853f: mov    DWORD PTR [ebp-0x4],0x5

8048530: mov    DWORD PTR [ebp-0x4],0x0
8048537: jmp    8048546 <_Z4blahi+0x22>

8048546: mov    eax,DWORD PTR [ebp-0x4]
8048549: leave
804854a: ret
```

Figure 2. A simple example of a control flow graph for an exported library function.

In most libraries, return codes are constants, typically defined with `#define` directives; these codes end up stored in a memory location or register (we refer to both of these as *locations*). For each assignment of a constant to a location, the static analyzer looks for the paths through the CFG that propagate this constant to the return location in an exit basic block. For most application binary interfaces (ABIs) the return value is placed in a well-defined location. For example, in the case of the Intel ABI, the return value is placed in the `eax` register, so we need to find the paths that propagate constants to the last `eax` write before a return instruction.

To make this path search efficient, the LFI profiler transforms the CFG $G(V,E)$, which has basic blocks $B_1, B_2, ...$ as vertices, into another graph $G'(V',E')$, with $V' = V \times \{l_1, l_2, ...\}$, where l_i are the locations to which constants are written. $E' = \{(<B_s, l_i>, <B_d, l_j>) \mid (B_s, B_d) \in E \wedge l_i$ *is propagated to* l_j *by* $B_s\}$. We say l_i is "propagated" to l_j by basic block B if the content of l_i is used to compute and write the contents of l_j within B. For calls to dependent functions, we consider all of the dependent function's return values (determined recursively) to be propagated.

The profiler identifies all writes to the return location preceding a return instruction and searches from this point in reverse, to find all paths in G' along which constants can be propagated to that location. One could think of this algorithm as a "reverse" constant propagation. Constant folding is not necessary, since compilers automatically do this when generating the library code, so the disassembled code offers no opportunity for further folding. We have not encountered any problems related to pointer aliasing in practice; it appears that modern compilers prefer to use the same canonical location to refer to these variables. To reduce the search space, the profiler generates G' on-demand, only expanding the nodes of interest.

In order to avoid injecting "obvious" non-faults, LFI can optionally apply two heuristics. First, it tries to distinguish success from error returns, to avoid injecting success returns; this heuristic removes 0-return values from all functions for which more than one constant return value were found (if only 0 was found, it is likely a null pointer return). Second, the LFI profiler eliminates short functions that return 1 or 0 and only check for conditions of the type `isFile()`; LFI infers that neither return value reflects a failure. Since both heuristics are unsound, they are disabled by default in LFI—we prefer to risk injecting some non-faults rather than miss valid faults.

A special type of dependency occurs in the C and C++ standard libraries (libc and libstdc++): they wrap kernel system calls, so many dependent functions reside in the kernel. LFI therefore performs static analysis on the kernel image as well, to identify the error codes that originate in the kernel and may be propagated by the libraries.

An alternative to the LFI approach is to obtain error return codes by parsing documentation. This approach has two main drawbacks. First, the analysis cannot be accurate, because documentation often uses natural language that is potentially confusing, such as *"the same errors that occur for link(2) can also occur for linkat()"* in the `linkat` man page, or *"returns 0 if successful, a positive error code otherwise"* in the `libxml2` documentation.

Second, documentation can be inconsistent. E.g., the `modify_ldt` man page claims three possible return values (`EFAULT`, `EINVAL` and `ENOSYS`), yet the LFI profiler found a fourth one (`ENOMEM`), confirmed through code inspection. We found similar inconsistencies in `libxml2`, where `htmlParseDocument` is alleged to only return 0 or -1 for success/failure, but it turns out it can also return 1 in some failure cases. Such disparities between documentation and reality can be the very source of program bugs, so an effective fault injection tool must be aware of them. LFI's fault profiles could be improved based on documentation, but this requires manual vetting.

It is possible to avoid analyzing exported library functions that the target program never calls, and thus save some profiling time. In LFI we opted to profile all exported functions of a library, because profiling is fast anyway (~20 seconds for the biggest libraries), and we wish to reuse profiles across multiple programs once they have been generated.

The LFI profiler is relatively portable: it obtains the exported symbols for a shared object file, disassembles it, and builds the control flow graph—these are steps that are performed using standard tools on most general-purpose platforms. The CFG analyses are independent of the ABI and platform features. As will be seen in §6.3, the LFI profiler currently works for three platforms: Linux/x86, Windows/x86 and Solaris/SPARC.

Limitations: Fault profiles may include false positives, i.e., return codes that can be returned by the corresponding function only when certain combinations of arguments are provided. For example, the `read` function in libc can return -1 and set `errno` to `EWOULDBLOCK` only when it is passed an asynchronous file descriptor. Inferring the relationship between arguments can be done using symbolic execution, but the current LFI prototype does not support this yet.

Indirect calls can pose a challenge to LFI's interprocedural constant propagation analysis. For such cases, the LFI controller could dynamically resolve indirect calls at runtime and inject the return codes corresponding to the function being called. However, prior work [17] and our own experience indicates that indirect calls are highly uncommon, even in event-driven object-oriented code. Moreover, our analysis of real libraries found that only 2.28% (758 out of 33,122) of indirect calls could actually affect the profiler's accuracy in static error code propagation.

In theory, indirect branches can make building the CFG hard. However, we analyzed 9,633 functions in 30 commonly used libraries and found that only 0.13% branches (104 out of 78,292) were indirect. The LFI prototype currently ignores the resulting CFG incompleteness.

LFI must be able to disassemble the libraries in order to analyze them; this may not work if the code is obfuscated. Fortunately, [17] reports that over 99% disassembly accuracy can be achieved in commercial grade applications. Since the LFI profiler and disassembler are loosely coupled, it is possible to use as good a disassembler as is available.

3.2. Side-Effects Analysis

Besides error return values, library functions may communicate to callers additional information regarding the encountered error, via channels such as output parameters, global variables, or thread local storage (TLS) variables, like `errno`. The LFI profiler automatically discovers and analyzes such side effects.

Shared libraries on most platforms consist of position-independent code (PIC), i.e., machine code that executes properly regardless of where it is loaded in memory. In PIC, all instructions referring to memory addresses use relative addressing. For example, in Linux, the `ebx/ecx` registers are loaded with the instruction pointer in the function prologue and subsequently used as a base address for accessing global or TLS variables. The LFI profiler starts out by finding the possible return codes and then it scans the basic blocks that contain the constant assignments, searching for possible writes to global/TLS variables. These writes are identified based on the use of the base address for computing the memory location to write to; propagating error codes to these locations is considered a side effect.

We illustrate with an example from GNU libc, where a function sets the `errno` TLS variable and places the return value in the `eax` register after a syscall returned an error:

```
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
```

Line 1 uses the standard PIC way of obtaining the current instruction pointer. In lines 2-4, it computes the address of the `errno` variable. Lines 5 and 6 compute the value to be stored in `errno` as the negative value of `eax`, in accordance with the Linux system call standard, and line 7 stores the value into `errno`. Finally, line 8 sets the return value of the function to -1. The profiler first finds line 8, then detects the side effect by analyzing the containing basic block; it concludes that exposing the error requires the injector to place -1 in `eax` and also set `errno` accordingly.

We take a similar approach for side effects reflected in output arguments, i.e., when the function writes to addresses passed in as arguments. Such output arguments are always found at a well known location—positive offsets from the base stack pointer when using frame pointers on the IA32 architecture, or stack/register combinations in general—so the LFI profiler detects writes to addresses obtained from such positive offsets. E.g., on IA32, we modified the algorithm used to detect possible return values such that it performs a forward search and looks for constant propagations to locations of the form `[ebp+??]`, instead of `eax`. If a chain of basic blocks that sets the return value also intersects a chain of blocks that propagates a constant to an `[ebp+??]` location, we consider it to be a side effect.

In order to understand how frequently the different ways of propagating error information are used in practice, we analyzed >20,000 functions in the Ubuntu Linux libraries. We used the ELSA C/C++ parser [6] to analyze the library headers in all the development packages, and combined this information with the LFI analyses described above. The results are summarized in Table 1—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 found that more than 90% of the exported functions in Linux shared libraries do not have side effects.

Return Type	**None**	**Error details in global location**	**Error details via arguments**
void	23.0%	0%	0%
scalar	56.5%	1%	3.5%
pointer	11.6%	1%	3.4%

Table 1. Statistics on how Linux libraries provide additional details on error conditions exposed to callers.

3.3. Fault Profile

The output of the LFI profiler is a fault profile of the analyzed library. This output is meant to be passed to the LFI controller, but it could also be used for other purposes, such as cross-checking API documentation. We therefore chose a general XML format that is both human-readable and easy to parse.

LFI generates one profile per analyzed library. For each exported function, the profile contains information regarding possible error return values, along with a specification of associated side effects for each such value. Here is a snippet of the profile generated for the libc `close` function:

```
<profile>
...
<function name="close">
<error-codes retval="-1">
  <side-effect type="TLS"
    module="libc.so.6" offset="12FFF4">
     -9
  </side-effect>
  <side-effect type="TLS"
    module="libc.so.6" offset="12FFF4">
     -5
  </side-effect>
  <side-effect type="TLS"
    module="libc.so.6" offset="12FFF4">
     -4
  </side-effect>
</error-codes>
</function>
...
</profile>
```

In case of error, `close` returns -1 and provides additional information via a TLS 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).

Incidentally, this is another example where man pages can be deceiving: on BSD systems, the man page accurately states that `close` can only set `errno` to `EBADF` or `EINTR`. On Linux, `EIO` is also possible, so programmers porting from BSD to Linux might forget to add a check for `EIO`; similarly, if porting to HP/UX they might forget to check for `ENOSPC`, or on Solaris they might forget about `ENOLINK`, all of which are return codes present in the corresponding libc libraries. LFI can automatically find the errors specific to the platform and test the programs with those values.

4. Fault Injection Scenarios

A fault injection scenario describes a sequence of faults to be injected; it can also be referred to as "faultload." Such a scenario pairs faults with *triggers*, i.e., conditions that, when true, should lead to an injection.

We designed a simple XML-based language to describe scenarios as sets of <trigger, fault> tuples. Every time a function is intercepted, the relevant triggers are evaluated and, if any is true, the associated fault(s) is/are injected. Due to space constraints, we do not describe the language in detail, but provide an illustrative example below:

```
<plan>
 <function name="readdir64" inject="5" retval="0"
           errno="EBADF" calloriginal="false" />
 <function name="readdir" inject="5" retval="0"
           errno="EBADF" calloriginal="false">
   <stacktrace>
     <frame>0xb824490</frame>
     <frame>refresh_files</frame>
   </stacktrace>
 </function>
 <function name="read" inject="20"
           calloriginal="true">
   <modify argument="3" op="sub" value="10" />
 </function>
...
</plan>
```

The first `<function ... />` trigger matches the 5th call to function `readdir64` and returns value 0 (null pointer), sets `errno` to `EBADF` ("bad file descriptor"), and does not call the original `readdir64` function. The second trigger matches the 5th call to function `readdir` and, if the call stack has the address `0xb824490` in the first frame and function `refresh_files` in the second frame, it injects `EBADF`. The third trigger matches the 20th call of the `read` function, modifies its 3rd argument (the number of bytes to be read) by subtracting 10 from it, and then passes the call on to the original `read` library call.

The current LFI prototype automatically generates two types of scenarios: exhaustive and random. In the exhaustive case, every exported function of every linked library is included, and consecutive calls to an exported function iterate through the possible error codes. In the random scenario, LFI takes as argument a probability, which is then used to randomly select both which call will return an error code and which particular error code it should return.

LFI allows testers to customize generated scenarios by using domain-specific knowledge: they might restrict the function calls to only a subset of interest, or specify a (partial) stack trace as a condition that must be matched by the runtime backtrace in order to trigger an injection. Scenarios can also be hand-written by testers, based on the library fault profiles. The random scenario is useful for quick-and-dirty testing, when testers have limited time and do not want to customize the fault scenarios, or when they lack specific knowledge that could target the testing better.

To help bootstrap fault injection testing experiments, LFI also comes with several ready-made fault scenarios for libc, such as all faults related to file I/O, all memory allocation faults, or all socket I/O faults.

5. LFI Controller

The LFI controller (Figure 3) receives from the profiler the fault profile(s) of interest along with a fault scenario, either automatically generated by the profiler or customized by the developer. It then generates interception stubs which are combined with boilerplate code to synthesize a new library. This synthetic library has the same API as the original one, but underneath this API encodes the fault injection logic. This new library is shimmed between the program being tested and the original library(ies).

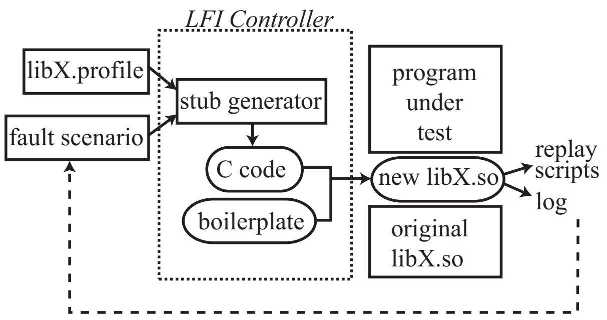


Figure 3. The LFI controller.

Once the stubs are generated and installed (§5.1), the LFI controller invokes a developer-provided script that starts the program under test, exercises it with the desired workload, and monitors its behavior to determine whether it terminates normally or with an error exit code. This information is collected in a log, along with an LFI-generated replay script for each fault injection test case; the test scripts allow the developer to diagnose and debug (see §5.2).

5.1. Interception Mechanics

The shimming of the synthesized library is system-specific. On Linux and Solaris, LFI uses the `LD_PRELOAD` environment variable to tell the dynamic linker to load the LFI-generated library before the original one. On Windows, LFI uses a combination of `WriteProcessMemory`/`CreateRemoteThread`, and passes the address of `LoadLibrary` as the thread start address, to force a process to load the synthetic library.

A synthesized library consists of stubs that intercept the library calls. Each stub determines the address of the original function, evaluates the triggers from the scenario and, if an injection is to be done, determines the return value/side effect to be injected and/or whether the call should be passed to the original function or not. If no injection is to be done, it cleans up the stack and jumps to the original function. In general, using a `jmp` instruction (instead of `call`) simplifies the handling of the original return address from the stack, because it avoids the need for save/restore. A stub looks approximately as follows:

```
int LIB_FUNC_NAME(void) {
  static void * (*original_fn_ptr)();
  static int call_count;
  call_count++;
  if (!original_fn_ptr)
    original_fn_ptr = (void* (*)())
                    dlsym(RTLD_NEXT,  #FUNC_NAME);
  if (eval_trigger(LIB_FUNC_NAME,
                  call_count, call_stack)) {
    /* determine return_code, side_effects */
    /* apply side_effects */
    return return_code;
  } else {
    /* return stack and registers to orig values */
    __asm__("jmp [original_fn_ptr]");
    /* orig func will return directly to caller */
  }
}
```

Interceptors for multiple libraries can coexist (see §6.4). This happens transparently, because the interception mechanism only relies on the function name, not on the library where the original function resides; thus, stubs for functions from different libraries do not interfere with each other.

Although interception is specific to the OS and CPU architecture, porting to new platforms is straightforward.

5.2. Controller Output

The LFI controller collects information that helps developers reproduce, understand, and fix the behaviors observed as a result of fault injection.

The *LFI log* is a text file that records each injection, the applied side effects, and the events that triggered that injection (e.g., call count, stack trace). This output can be used to match injections to observed program behavior, as well as to refine the fault scenario.

The *replay scripts* are automatically-generated XML files that can be fed back to the LFI controller to reproduce the desired test case on a subsequent run. Replay is not always 100% accurate, because LFI does not control thread interleaving, timer inputs, etc. We have found that these replay scripts can save a lot of time during ad-hoc testing, and can also augment existing regression test suites.

6. Evaluation

Our goal in building LFI was to make testing based on fault injection easier and less human-intensive; in this section we evaluate the extent to which we reached this goal. We analyze the effectiveness of using LFI in testing (§6.1), measure its efficiency (§6.2), assess the accuracy of the LFI profiler (§6.3), and measure the performance overhead incurred during testing (§6.4). LFI currently works on Linux and Windows (both Intel IA-32 and IA-64 architectures), as well as Solaris (SPARC architecture). We expect LFI to be easily ported to other systems as well.

6.1. Effectiveness

Ease of Use: LFI's primary contribution is ease of use. The human effort involved in the basic use of LFI is small: it requires issuing two commands, one for profiling and one for running the tests. When the tester modifies the default fault profiles, more time is required, but we expect this to be easier than directly scripting fault injection experiments. Below, we illustrate LFI usage with an example.

We tested Pidgin [15], a popular instant messenger client, by instructing the LFI controller to launch it and exercise a random fault injection scenario on I/O functions with 10% probability. Shortly after we entered the IM login details in Pidgin, it crashed with a `SIGABRT`.

We restarted Pidgin using the corresponding replay script and attached with gdb; it crashed again and we were able to inspect the program state. In a matter of minutes, we discovered the issue: Pidgin forks a DNS resolver process to perform host resolution asynchronously; this process then communicates back to the parent via a pipe. The child does not handle the case when writes fail or are incomplete. As a result, the child may write the answer to the parent, but, if the write is incomplete, it may subsequently write additional data corresponding to another request. As a result, the parent reads the status (which is ok), and then reads the size of the (resolved) address—due to the partial write, this read returns data written after the injection, in our case a very large value. The parent calls `malloc` for this amount of memory, which results in `SIGABRT`, because it is unable to allocate the memory. Further details appear in the bug report we filed [16].

Improving Coverage: Besides ease-of-use, effectiveness can also be measured by whether LFI can improve existing regression test suites. We considered the MySQL database server, which is the most mature open-source RDBMS, first released in 1995; it claims 11 million installations [4]. MySQL ships with its own thorough test suite. The MySQL 5.0 test suite achieves 73% overall basic block coverage, which is remarkable for an open-source project; we therefore set out to see if we can improve this with no human effort.

We ran LFI in fully automatic mode, generating a random fault injection scenario based on libc. With no human help, LFI improved the coverage of the MySQL test suite to at least 74% overall; in some modules (such as the InnoDB `ibuf` implementation) coverage improved by 12%. We expect the coverage numbers to be slightly higher, because in 12 cases MySQL crashed with `SIGSEGV` and the coverage information for those test cases was not saved. We are encouraged by the fact that, with no human assistance, LFI was able to improve a mature, extensive test suite.

Finding Obscure Scenarios: A third aspect of effectiveness is whether LFI is able to exercise scenarios that existing tools would not find. As already mentioned in §3.1 and §3.3, the LFI profiler found several return error codes that are missing from the API documentation of popular libraries. By analyzing directly the binary, LFI helps testers find fault scenarios that they would otherwise not be aware of; knowing these additional fault scenarios enables testers to write more thorough tests.

6.2. Efficiency

The running time of a test tool is an important factor in its adoption, because testers are generally unwilling to wait long for results. For example, the long running times of model checkers have discouraged their wide use in testing. The LFI profiler is fast: we measured profiling time 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 large library (libxml2, with 1612 exported functions and a 897 KB code segment). To profile the >1,000 libraries found on a typical Linux system takes several hours, but in practice we expect testers to only profile the libraries used by the program of interest. When updating a library on the system, which we expect will happen about once a month, it takes on the order of minutes to re-analyze the updated library and its dependencies in order to update the library fault profile.

Profiling time is mainly influenced by code size (i.e., number of machine instructions). The number of hops in the propagation of return codes to the `eax` (or equivalent) register also has an impact, but we have found this number to be always 3 or less, due to compiler optimizations like constant propagation and constant folding, so its effect is negligible.

6.3. Accuracy

Accuracy of the profiler can be expressed as TP/(TP+FN+FP), i.e., the ratio of true positives to the sum of true positives, false negatives, and false positives. A true positive is an error return code that was correctly found; a false negative is a returnable error that was not found; a false positive is a reported error code that cannot actually be returned. A factor that can influence accuracy is the number of indirect branches and indirect calls, because, as explained in §3.1, indirection poses a challenge to the static analyzer. Accuracy is also 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.

For evaluation purposes, the "ground truth" for deciding what is a false vs. true positive or negative cannot be easily determined, because written documentation is not reliable; if we wanted precise numbers, labor-intensive manual code inspection would be required. We performed such an analysis on a small library (libpcre, with 20 exported functions) and found the accuracy to be 84% (52 true positives, 10 false negatives, 0 false positives).

To scale the evaluation, we considered 18 additional libraries on 3 platforms, but this time we considered the ground truth to be the documentation. We wrote documentation parsers for each of the measured libraries. While this evaluation is inexact, it is the only practical method of comparison. In Table 2 we show the results of running the LFI profiler on the respective binaries.

With no access to documentation, source code, or human assistance, the LFI profiler achieves on the order of 80%-90% accuracy. False negatives result in missed fault scenarios, while false positives result in time wasted by the developer verifying that the injected fault condition cannot actually occur in practice. Should structured documentation exist and a documentation parser be available, it can be combined with LFI's static analysis to yield higher accuracy.

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

Table 2. Profiler accuracy with no human assistance, no documentation, and no source code, on Linux/x86, Solaris/SPARC, and Windows/x86. We show true positive (TPs), false negatives (FNs), and false positives (FPs) relative to library documentation.

6.4. Performance Overhead

The final question we wish to address is whether the process of injecting library-level faults slows down the system to the point that its behavior is no longer representative. If this was the case, the value of testing would be decreased.

We measured the overhead introduced by the LFI controller in the AB [2] benchmark on the Apache httpd server, while LFI was simultaneously performing fault injection on the calls to GNU libc, libapr, and libaprutil. GNU libc is a large library, with 1535 exported functions, while the two libraries comprising the Apache Portable Runtime (APR) are medium-sized, totaling a little over 1,000 functions.

We allowed LFI to produce a random fault injection plan with 10 triggers on the top-10-most-called functions in Apache httpd, 100 triggers on the top-100, 500 triggers on the top-300, and 1,000 triggers on the top-300, respectively (in the last two cases, there were multiple triggers for the same function, corresponding to different error returns). In these experiments, LFI always passes the call through to the original library after evaluating the trigger, in order to allow Apache to properly complete the benchmark. In each test we ran 1,000 requests with AB.

Table 3 summarizes two sets of results, obtained with two different workloads: static HTML and PHP. The latter is more dynamic and performs many more library calls than the former, which implies that the triggers have to be evaluated considerably more times. As can be seen, the overheads introduced by trigger evaluation are negligible.

	Static HTML	**PHP**
Baseline (no LFI)	0.151 sec	1.51 sec
10 triggers	0.156 sec	1.53 sec
100 triggers	0.156 sec	1.53 sec
500 triggers	0.158 sec	1.57 sec
1,000 triggers	0.159 sec	1.60 sec

Table 3. Runtime overhead of using LFI in the Apache httpd server with three simultaneous libraries (GNU libc, libapr, and libaprutil). We report completion time of 1,000 AB requests. The baseline represents Apache httpd without any interference from LFI.

We also ran the SysBench [23] Online Transaction Processing (OLTP) benchmark on the MySQL RDBMS with LFI applied to GNU libc; we varied the number of triggers from 10 to 1,000. Table 4 shows the results for two different workloads: read-only and read-write queries.

	Read-only	**Read/Write**
Baseline (no LFI)	465.28 txns/sec	112.62 txns/sec
10 triggers	464.48 txns/sec	112.08 txns/sec
100 triggers	463.19 txns/sec	111.53 txns/sec
500 triggers	460.80 txns/sec	110.88 txns/sec
1000 triggers	459.39 txns/sec	110.10 txns/sec

Table 4. Runtime overhead while applying LFI to the MySQL database server. We report number of transactions per second, as reported by SysBench OLTP.

As in the Apache case, the runtime overhead during testing is negligible, even for a large number of triggers. It is apparent that overhead is influenced both by how intensely the program uses the profiled library and how many triggers are present in the fault injection plan.

7. Related Work

Performing fault injection at the software level is attractive, because it does not require expensive hardware mechanisms, and it can be used to target various layers in the software stack. Software fault injectors can either be inserted directly into applications, or can be shimmed between existing layers of the software stack.

Software fault injection has seen varied uses in the literature, ranging from use as a method for testing the robustness of device drivers [1] to testing general-purpose operating systems [9, 13, 11, 12, 10, 3] to mission-critical systems and real-time systems [18, 19].

NFTAPE [20] is an example of a fault injection framework that can inject various low-level faults with the main purpose of assessing dependability of distributed systems. In our experience, establishing the mapping between low-level faults and higher-level application events across several layers of the software stack is not easy, making diagnosis and debugging tedious.

Our work is focused on library-level fault injection, because we view this as an ideal layer for doing realistic testing: it is the interface that is most likely to expose applications to failures that occur in their environment.

Work related to this idea includes Ballista [14], an early system for testing the robustness of a library or operating system API by passing boundary values as arguments. It relies on domain-specific knowledge to select arguments that will stress the tested component and also needs access to the corresponding function prototypes. Similarly, HEALERS [7] searches for arguments that can cause a library function to crash; it then generates wrappers that protect the vulnerable functions from the pathogenic arguments.

Our work operates in the opposite direction: we test the application by giving it error return values from the library. This way, we verify that the program reacts properly to the exposed error conditions, e.g., check how it handles situations when `malloc` is unable to allocate memory.

Research interest in this type of library-level fault injection is relatively recent and, to our knowledge, debuted with FIG [5], a tool used to verify the recovery mechanisms of applications that use the GNU libc (glibc) library. FIG injects faults solely in calls to glibc and requires that the injectable glibc errors be hardcoded. In contrast, LFI can be used with any library and automatically generates stubs that perform complete fault injection, including side effects. We also offer control over the injection process via an XML-based fault description language to flexibly specify injection scenarios.

Süßkraut & Fetzer [21] introduced a system that finds application problems via library-level fault injection and then patches the applications to protect against these faults. The system is limited to libc and relies on man pages to determine possible error return values. As shown in §3.1 and §3.3, man pages can sometimes be incorrect, so in LFI we extend the man page parsing approach with static analysis of the library binaries, to automatically extract error return codes. [21] also requires information on the function prototypes in the form of header files to generate corresponding wrappers and uses systematic error injection. LFI eliminates the need for header files and decouples the specification of fault scenarios from the fault injection mechanism, thus allowing for more flexible test scenarios (systematic, random, custom, etc.).

Süßkraut & Fetzer [22] further introduced a technique for learning library-level error return values by injecting system call errors (i.e., faults at the boundary between the operating system and the library) and observing their propagation to the libc interface. The LFI profiler uses static analysis of binaries, because the system call injection approach is limited to libc (the only library that directly accesses the system call interface), and it requires recompiling the kernel, in order to export the system call table. We believe that direct analysis of the binaries makes LFI more widely applicable.

8. Conclusion

We presented LFI, a tool for making fault injection-based testing more efficient and accessible to developers and testers. LFI injects faults at the boundary between shared libraries and target programs, in order to verify whether the programs correctly handle failures exposed by the libraries. LFI automatically extracts information from the binary libraries regarding possible error return codes and their side effects. Based on this fault profile, LFI generates various fault injection scenarios, which can be used directly or modified as desired by testers. Based on the fault profile and scenario, LFI synthesizes a shim library that injects the desired faults and records the behavior of the target program.

LFI generalizes to the shared libraries found on common Linux, Windows, and Solaris systems, and profiling takes on the order of seconds for each library—this makes it practical for use in real development. We have shown that LFI is useful even when run without human assistance and no access to documentation or source code—it was able to increase test coverage even on the extensive MySQL test suite, by exercising recovery code paths that are not touched by regular testing. The performance overhead incurred during fault injection is negligible, which means that program behavior remains realistic during testing.

9. Acknowledgments

We thank Ankit Singla for his early contributions to LFI. We are grateful to the anonymous reviewers and Ming Yu and Liviu Ciortea for their help in improving this paper.

References

[1] A. Albinet, J. Arlat, and J.-C. Fabre. Characterization of the impact of faulty drivers on the robustness of the linux kernel. In *Intl. Conf. on Dependable Systems and Networks*, 2004.

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

[3] J. Arlat, J.-C. Fabre, M. Rodríguez, and F. Salles. Dependability of COTS microkernel-based systems. *IEEE Trans. Comput.*, 51(2), 2002.

[4] C. Babcock. Sun locks up MySQL, looks to future Web development. InformationWeek. Retrieved on 2008-02-27. http://informationweek.com/news/showArticle.jhtml?articleID=206900327.

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

[6] ELSA. http://www.eecs.berkeley.edu/ smcpeak/elkhound/sources/elsa/. Accessed on 15-Mar-2009.

[7] C. Fetzer and Z. Xiao. HEALERS: A toolkit for enhancing the robustness and security of existing applications. In *Intl. Conf. on Dependable Systems and Networks*, 2003.

[8] http://ftp.gnu.org/gnu/glibc/. Accessed on 15-Mar-2009.

[9] T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun. Experimental analysis of the errors induced into linux by three fault injection techniques. In *Intl. Conf. on Dependable Systems and Networks*, 2002.

[10] D. Joao and M. Henrique. Multidimensional characterization of the impact of faulty drivers on the operating systems behavior. *IEICE Trans. Info. and Sys.*, 86(12), 2003.

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

[12] A. Johansson, N. Suri, and B. Murphy. On the selection of error model(s) for OS robustness evaluation. In *Intl. Conf. on Dependable Systems and Networks*, 2007.

[13] K. Kanoun, Y. Crouzet, A. Kalakech, A.-E. Rugina, and P. Rumeau. Benchmarking the dependability of windows and linux using postmark workloads. In *Intl. Symp. on Software Reliability Engineering*, 2005.

[14] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. In *Intl. Symp. on Software Reliability Engineering*, 1997.

[15] Pidgin. http://www.pidgin.im.

[16] Pidgin - ticket 8672. http://developer.pidgin.im/ticket/8672.

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

[18] M. Rodriguez, J. Arlat, and J.-C. Fabre. Building SWIFI tools from temporal logic specifications. In *Intl. Conf. on Dependable Systems and Networks*, 2003.

[19] V. Sieh, O. Tschache, and F. Balbach. VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions. In *Intl. Symp. on Fault-Tolerant Computing*, 1997.

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

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

[22] M. Süßkraut and C. Fetzer. Learning library-level error return values from syscall error injection. In *European Dependable Computing Conference*, 2006.

[23] Sysbench. http://sysbench.sourceforge.net, 2008.