Studying Application–Library Interaction and Behavior with LibTrac

Eric Bisolfati, Paul Marinescu, and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

LibTrac is a tool for studying the program/library boundary and answering questions like: Which library functions are called most often? Are there library usage patterns that distinguish one class of applications from the others? Do programs generally retry failed I/O calls or not?

The answers to these questions are essential to anyone employing library-level fault injection [1, 6] in software testing. On the one hand, the program-library boundary is an appealing location for injecting faults, because the cost of doing so is low, and one can emulate a wide range of real-world failures. On the other hand, developers must decide a priori which library calls to fail, when, and in what way. The space of possibilities is vast, so developers need tools like LibTrac to make informed choices for test scenarios.

We used LibTrac to study 13 real-world systems; we report here some of the results. Compared to existing library tracers, LibTrac incurs one to two orders of magnitude less overhead, thus offering considerably more realistic study conditions.

1 Introduction

Shared libraries are used heavily by programs, because they offer modularity and opportunities for code reuse. Although the shared library infrastructure requires cooperation of the compiler, linker, and runtime linker/loader, this is all largely transparent to developers. As a result, virtually all Linux programs use at least the standard C shared library, while Windows programs use at least services provided by the *kernel32.dll* shared library.

Even though shared libraries are widely used, little is known about how programs use them or how they react to library-level failures. Questions like how does a database server handle I/O errors, which library functions are always called together, or which functions are used the most by instant messaging clients can offer insights useful in testing and optimization. These questions arise especially when employing library-level fault injection for software testing [1, 6].

Some tools [3, 8, 11] can retrieve library function call information, but they are not aimed at testing-related tasks and do not offer the necessary level of detail. Existing library tracers [5] are typically heavyweight and too slow for drawing representative conclusions about application–library interaction and behavior. This is why we developed LibTrac, a library tracer that is an order of magnitude faster than ltrace [5] and offers considerably richer information, such as second-order library calls (i.e., a library calling another library), rich statistics, etc.

LibTrac collects data and computes statistics about the program-library interface to help understand program behavior. LibTrac retrieves at runtime all details about function calls, call site addresses, function parameters, and return values, while keeping overhead low. Systems like Microsoft Detours [4] offer an API for intercepting library functions, but do not include the logic for automatically setting up interception or gathering the data and processing it.

By using LibTrac on 13 real-world programs, we gained unexpected knowledge about entire classes of applications. For example, instant messengers tend to use shared libraries in similar ways: as shown in §3.2, Pidgin and XChat, even when used on different instant messaging networks, exhibit strikingly similar library call distributions, despite the fact that they do not share any code. This suggests that a library-level fault injection scenario built for Pidgin could be productively reused for other instant messengers.

In the rest of the paper, we present the design and implementation of LibTrac (§2), show results for several different applications (§3), briefly review related work (§4), and conclude (§5).

2 Design and Implementation

We now present LibTrac's interceptor-based design (§2.1), the techniques for ensuring that the interception layer can safely coexist with the traced target in the same address space (§2.2), and LibTrac's optimizations aimed at minimizing interference with the target program (§2.3). Finally, we describe a LibTrac prototype for Linux (§2.4).

Appears in Proceedings of the Intl. Conf. on Dependable Systems and Networks (DSN), Chicago, IL, June 2010

2.1 Interception of Library Calls

LibTrac interposes between the program of interest and the shared libraries that the program uses. LibTrac uses the same mechanism as LFI [6], i.e., it automatically generates stubs for each intercepted function, without requiring access to source code, debug symbols, or documentation.

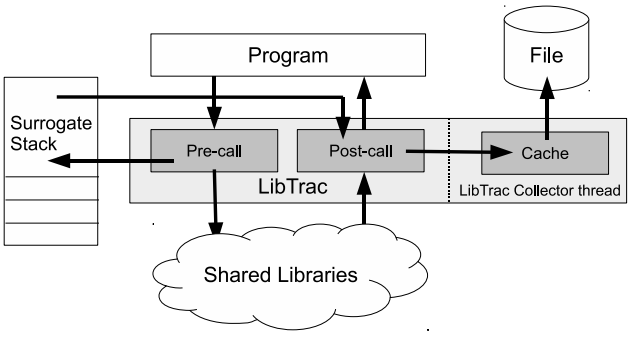

Figure 1. LibTrac architecture.

Every time a library function call is intercepted, LibTrac executes before and after calling the original function. We refer to the processing done by LibTrac before the call to the shared library as the *pre-call*, and to the processing occurring after the call as the *post-call*.

When intercepting a function call, the pre-call has to first save the state of the program, so as to keep the interception invisible to the hooked function (e.g., no registers may be altered by LibTrac). Since the number and the types of the arguments are not known, the pre-call must save *all* registers; this is done before starting processing, and registers are restored before calling the original function. This implies that LibTrac stubs may not store anything on the stack across the original call, since the stack pointer must be restored as well. LibTrac employs a surrogate stack (see Figure 1), which is also used to handle nested functions.

In the pre-call, LibTrac retrieves and logs the name of the called library function, its arguments, the order of the call (i.e., the number of intercepted library functions currently on the stack), and the call site address. Before executing the original function, the return address is modified to ensure control is returned to the stub after the original returns.

After executing the original library function, the post-call stage is executed, where LibTrac saves the value returned by the original call and performs various housekeeping before returning control back to the traced program: it creates a new log file if `fork` was called, and flushes the log collector thread's buffers to disk, if they are full.

LibTrac heuristically determines the number of arguments each library function takes by statically analyzing the program's binary: for each call, LibTrac checks which registers are modified in the instructions preceding the call, each corresponding to an argument according to the architecture-specific calling convention. The final number of arguments is obtained by taking the most often encountered result over all the calls to the same function. To measure the accuracy of this heuristic, we compared the documentation of the library function with the number inferred by LibTrac—LibTrac got the number of parameters right 96% of the time. Over half of the remaining 4% errors were due to variadic functions like `printf`; the rest were induced by compiler optimizations not yet handled by LibTrac.

2.2 LibTrac – Library Coexistence

When intercepting library calls, two classes of issues arise. First, the classic method to obtain the address of the original function in an interceptor would be to invoke the dynamic linker (e.g., the `dlsym` function in Unix). Alas, this can result in an infinite loop when intercepting functions called by `dlsym` itself. To circumvent this problem, LibTrac computes the address where each library function is loaded by analyzing the memory image of the executable and computing the actual address by adding the offset of each function to the base address of its library.

Second, programs like Web browsers may decide late in their execution which libraries to use. For instance, for each plugin, Firefox on Linux uses the `dlopen` function to dynamically load its associated library only when the plugin is first invoked. Since LibTrac cannot determine a priori which shared libraries will be used, it must intercept each `dlopen` call and analyze the arguments in order to determine the name of the external libraries being used.

2.3 Minimizing Interference

LibTrac's first design goal is to never alter the behavior of the traced program. This, however, limits the flexibility of the interceptors. For example, LibTrac can only use library functions that are reentrant, because LibTrac calling non-reentrant functions would affect the traced program's subsequent calls to that same function (e.g., `strtok`).

Another design goal is to reduce performance interference, because introducing delays in the target program can alter indirectly its behavior. Such performance interference can be indeed substantial in the case of existing tools, as we show in Table 1: we report the time it takes the Apache Web server to serve 1,000 HTTP GET requests and the Emacs text editor to indent a C source file in batch mode.

System	**No tracing**	**LibTrac**	**Callgrind**	**ltrace**
Apache	0.2 sec	0.9 sec	3.9 sec	42.0 sec
Emacs	0.2 sec	0.5 sec	14.0 sec	61.0 sec

Table 1. Performance interference of library tracers.

The two main bottlenecks in library tracers are context switches and/or I/O operations that record data to disk. To avoid these bottlenecks, LibTrac writes the collected data asynchronously to disk, via a dedicated I/O thread, thus also avoiding context switches. LibTrac also does not suffer from `Callgrind`'s performance penalty imposed by running the target application on a virtual processor. Finally, compared to statistical profilers like Intel VTune [12], LibTrac has a similarly low overhead, but has the advantage of collecting *all* the information regarding library calls—not just samples—along with all their details.

2.4 Implementation

Our LibTrac prototype runs on Linux as a shim library loaded via the `LD_PRELOAD` mechanism, thus being able to run in the same address space as the traced program. Even though this mechanism is Unix-specific, LibTrac's design can be applied to other operating systems. For example, on Windows one can use the tools analogous to *ldd* and *readelf*, while the interception itself can be implemented using the Microsoft Detours [4] library. Interception is by necessity specific to the OS and CPU architecture, but all the data analysis algorithms are generic.

Due to the `LD_PRELOAD`-based interception, LibTrac can only collect data regarding functions in shared libraries. Most libraries are shared, so LibTrac addresses the common case. Nevertheless, LibTrac could use binary instrumentation to also intercept calls to statically linked libraries, while keeping the overall design the same.

3 Using LibTrac in Practice

In this section we show how LibTrac can be used to detect patterns of behavior in 13 real-world applications; these patterns can then be used to improve the efficiency of testing, especially fault injection. We show in Table 2 the target applications; the last column (imported functions) indicates the number of library functions used by each application.

We traced the target executables using LibTrac, while applying a workload specific to the type of application. For each library function call, LibTrac records its order, call site, arguments, and the return and `errno` values. LibTrac's offline component processes the recorded data and computes statistics.

For databases, we used the TPC-W and TPC-C benchmarks [9]. For browsers, we used Google's V8 benchmark suite, which exercises the browser's JavaScript capabilities [10]. For instant messengers and text editors, we hand-crafted a representative workload. All programs were evaluated under two x86_64 Linux distributions (Archlinux 2009.08 and Ubuntu 9.04), running on a quad-core 2GHz Intel Xeon processor with 4GB of RAM.

Class	Program	Size (KB)	Imported functions
Databases	MySQL 5.1.41	8806	248
	SQLite 3.6.20	51	118
	PostgreSQL 8.4.1	4505	282
Web browsers	Epiphany 2.28.1	84	1508
	Firefox 3.5.5	88	54
	Seamonkey 2.0.4	17500	2027
Instant messengers	BitchX 1.1	1433	153
	XChat 2.8.6	632	782
	Pidgin 2.6.4	923	1789
Text editors	emacs 23.1	11264	753
	gedit 2.28.2	659	1245
	scite 2.01	1843	519
	nano 2.0.9	165	135

Table 2. Applications used in our study.

In the rest of this section, we provide a few data points on how applications behave when they encounter naturally occurring failures, how often this happens, and how we can leverage this information (§3.1). Afterward, we explain how non-failure behavior can also provide valuable information for testing, in particular for fault injection (§3.2). Additional data can be found at `http://lfi.epfl.ch/`.

3.1 Behavior in the Face of Failure

In this paper, we say a library function "fails" when it returns an error value (e.g., a NULL pointer) and potentially sets the `errno` variable. However, LibTrac allows specifying custom failure-detection criteria on a per-function basis, which offers far more control than shown here.

Interestingly, LibTrac detected function call failure in all analyzed programs, even though there were no underlying failures. This shows that error recovery code should be expected to be exercised even under "normal" operating conditions. Moreover, in our experiments, when one failure occurred, it typically triggered cascading failures; these were generally well handled by the applications we studied.

We determined the number of failed calls to each shared library function and compared it to the total number of calls to that function and to the total number of intercepted calls. This directly provides information about function failure modes. It can be used, for instance, to do fault injection experiments with a realistic distribution of faults, i.e., one that has similar statistical characteristics to observed real behavior. Moreover, identifying ways in which functions fail in some parts of the program enables testers to inject similar failure behavior in other parts of that program where failures may not normally occur.

In the applications we analyzed, the `read` function call was the one that set `errno` the most often. In fact, the `read` function can fail in more than a third of its invoca-

tions. In Table 3 we show sample data for gedit; the "failure rate" column indicates what fraction of the corresponding calls resulted in failure. The most frequently set value of `errno` is `EAGAIN`, indicating "resource temporarily unavailable." This means that a file descriptor was marked for non-blocking I/O, and no data was ready to be read at the time when the `read` call was made.

Name	Failures	Failure rate
read	901	33%
open	191	25%
fopen	68	74%

Table 3. Top 3 most often failing library function calls in the gedit text processor.

Which functions have the highest failure rate is strongly dependent on the class of applications. Table 4 shows results typical of instant messaging clients: file and network-related function failures are quite common. Note that a failed call to `close` can result in data loss, if the data written to the corresponding file descriptor is not flushed.

Name	Failures	Failure rate
access	56	86%
close	1	33%
connect	1	50%

Table 4. Top 3 most often failing functions in the BitchX instant messaging client.

We found that usually less than 0.2% of all library calls result in failure. Yet, among these 0.2%, there exist functions that are called only once and fail (i.e., a failure rate of 100% for that function). We suspect this has the effect of disabling certain features in the program; for instance, a failure to open a file with write access causes gedit to completely disable the "save" functionality for that file.

LibTrac can detect if, upon encountering a failed library call, the target program retries the call. For this, LibTrac uses an offline heuristic analysis: if two calls are made "one after the other" from the same location, and the first failed, LibTrac concludes that the second one was a retry. We define "one after the other" in terms of the number N of intercepted calls separating the two calls. The value of N was computed by starting from a small value and gradually increasing it until the heuristic started producing false positives. We found N=10,000 calls to be suitable for most applications with a GUI and generally for applications with deep call stacks; smaller values offer better accuracy for other applications (e.g., console-based ones). We manually analyzed the accuracy of this heuristic for gedit, and Table 5 summarizes the results.

Name	Correctly detected	Not detected
read	840	22
open	185	2
fopen	56	3

Table 5. Number of detected and missed function retries for gedit, using LibTrac's offline heuristic.

In the case of editors or text processors, like gedit, more than half of the `read` retries eventually succeed.

3.2 Behavior under Success Conditions

Interesting observations can be made even when looking at successful operation. For example, for practically all applications, the number of times a library function is called drops quickly: a difference of more than 40% is common between the first and the second most called functions, as illustrated for Pidgin in Figure 2. The number of times a function is called is an important consideration when deciding which is the most "promising" candidate function for fault injection.

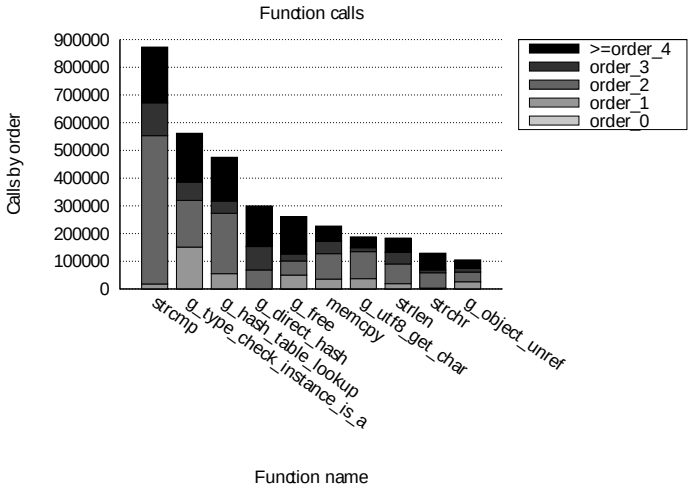

Figure 2. Top 10 most called functions in Pidgin, with their respective order (i.e., number of other library calls on the stack at the time of the considered call).

In general, the deeper a fault occurs or is injected, the higher the probability that the fault does not propagate to the application. In other words, injecting an error in an order-0 function is more likely to find application bugs than injecting in an order-10 function—in the latter case, there are more layers that could mask the injected fault and prevent its propagation to the application. In fact, injecting at a high

order will more likely test the library than the application.

The order average, however, is not sufficient to decide on how to construct a test scenario—the variance of the order is important as well. In Table 6 we show the average function order and its variance for applications with graphical user interfaces, like instant messengers and editors. Usually, GUI functions are called as zeroth order functions, and they in turn call other library functions; in terms of fault injection, chances are that injected faults will be handled directly by the graphics library. We see that a high average call order is associated with a high variance, which suggests that faults injected in high-order functions still have a fairly high chance of propagating to the applications.

Program	Average	Variance
Pidgin	3.26	5.07
XChat	2.44	3.04
BitchX	0.62	0.24
Firefox	2.81	5.07
Seamonkey	2.46	5.23
Epiphany	2.38	4.11

Table 6. Library function call-order average and variance for instant messengers and browsers.

In the remainder of this section, we present a few interesting characteristics of individual classes of applications, as revealed by LibTrac.

Instant Messengers: Despite IM clients being network applications, we found that string-related functions (not network functions) are the ones called most often. Such functions are not particularly interesting for library-level fault injection, because their outcome depends solely on their arguments and, when they fail, they do not return error values. However, this dominance of string functions can point the test engineer in another direction: the large amount of text processing done by such applications can make them a good target for fuzz testing [7], a technique which aims to find bugs by providing random or malformed inputs.

The top-10 most called library functions for Pidgin and XChat are quite similar (Figures 2 and 3), even though they do not share the same code and were tested on different instant messaging networks. There is however a difference in their call orders. The distribution of the locations of call sites (places from where a function is called) is similar as well, suggesting perhaps that the applications' design is similar. Testing scenarios that are effective for one of the programs may offer good results for the other as well.

Browsers: Firefox and Seamonkey exhibited similar library usage, including average call order and variance. In this case, the results were expected, since the applications share code by using the same HTML rendering engine.

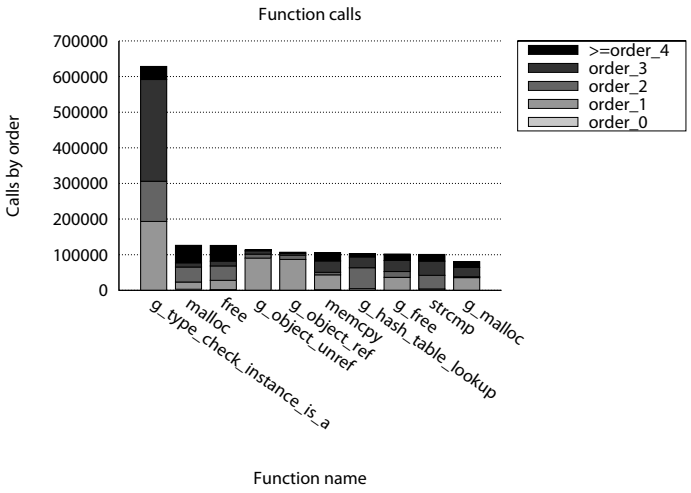

Figure 3. Top 10 functions called for XChat, with their respective order.

Text Editors: The editors we analyzed use mostly string processing functions, memory allocation functions, and calls to the C standard library.

In the case of Emacs, however, we noticed substantially fewer library calls than in any other editor; in some cases there were an order of magnitude fewer for the same workload (e.g., much fewer calls to `strcmp` than in gedit). We suspect this is because Emacs, being written in Lisp, uses a large number of internal functions and does not rely on external libraries, thus evading LibTrac's interception. Figure 4 shows the top-10 most called functions seen by LibTrac, ordered by the number of different call sites.

Databases: We found memory management and memory manipulation functions to be the most frequently called

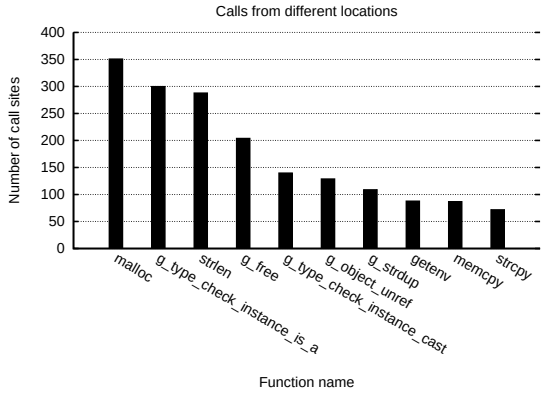

Figure 4. Top 10 most often called library functions in Emacs, ordered by number of call sites.

ones in database systems. For example, the `memcpy` function is the most often called in MySQL, and it appears in the top-10 for both PostgreSQL and SQLite. In the case of SQLite, we observed a large number of memory allocation calls compared to the other database systems, suggesting that SQLite does not use a custom memory allocator.

We sometimes observed imbalances in functions that normally appear in pairs. For example, `pthread_mutex_unlock` is sometimes called more often than `pthread_mutex_lock`. The same applies for the `free`/`malloc` pair, where `free` is called more often than `malloc`. The extra calls to `free` receive a `NULL` argument, which turns into a no-op. Perhaps this is a design choice that chooses code readability over squeezing out the last ounce of performance.

Good injection scenarios can be suggested by these imbalances reported by LibTrac. For example, a mutex must be unlocked regardless of whether its holder terminates successfully or not. Yet, a seldom-occurring error recovery path may miss the unlock instruction. Therefore, it can be of interest to inject errors while the mutex is held, and to verify that it is eventually properly released.

Another common pairing is `access` and `open`. The `access` function is used to verify that the specified file can be opened with the desired attributes, as in:

```
if (access(file, R_OK) != 0) {
   exit(1);
}
f = fopen(file, "r");
```

This code pattern constitutes a "time of check to time of use" bug when executed by a setuid program. A race condition can occur, in which another process modifies the file between the call to `access` and the call to `open`. This can lead to a compromise of the system if an attacker can replace the original file with a symbolic link to another (restricted) file, which the setuid program will unknowingly open and overwriting or disclose to unauthorized parties.

4 Related Work

Using library interposition for analyzing programs is not new: historically, it has found various uses in research and industry, ranging from record/replay to profiling [2]. Intercepting library calls is also done in the standard Linux library tracer, ltrace [5], but it introduces large overhead because the interception uses breakpoints. Performance improvements were made by [3] using binary module injection and function redirection, but still the overhead is high. Of course, this offers more flexibility, by allowing the tracing of both internal functions and library calls, but the high overhead can impact the representativeness of observed application behavior.

In the context of this prior work, LibTrac's contribution is a way to trace library calls with low overhead. Furthermore, none of the existing tools can analyze program behavior from an error handling point of view—most tools offer only limited information that is useful for testing.

5 Conclusion

We presented LibTrac, a tool that analyzes the interaction between applications and shared libraries with the main purpose of providing useful information for testing. LibTrac has an order of magnitude less overhead than other library tracers and answers important questions related to library function usage patterns. We evaluated LibTrac on four classes of applications, totaling 13 real-world programs, and presented here a subset of the facts we learned about these applications.

Acknowledgments

We would like to thank Christof Fetzer, the anonymous DSN reviewers, and our EPFL colleagues for their invaluable help in improving this paper.

References

[1] 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.

[2] T. W. Curry. Profiling and tracing dynamic library usage via interposition. In *USENIX Summer Technical Conf.*, 1994.

[3] The Embedded ELF Tracer: Etrace. http://www.eresi-project.org/wiki/TheEmbeddedELFtracer.

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

[5] ltrace. http://freshmeat.net/projects/ltrace/.

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

[7] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[8] strace. http://sourceforge.net/projects/strace/.

[9] The TPC-C OLTP benchmark. http://www.tpc.org/tpcc.

[10] Google's open source JavaScript engine. http://v8.googlecode.com/svn/data/benchmarks/current.

[11] Valgrind. http://valgrind.org/.

[12] Intel VTune Performance Analyzer. http://software.intel.com/en-us/intel-vtune/.