LibSE: Inferring return values and path conditions for library functions

Paul Marinescu
Advanced Software Analasys and Verification
EPFL
Lausanne, Switzerland

Abstract

Determining the possible ways in which a certain software component can fail is a critical yet overlooked aspect in the development and testing of general-purpose software, partially because there are no easy-to-use tools that would make this information available to the tester. In this paper we present LibSE (Library Symbolic Execution), a tool for automatically inferring the possible modes of failure for one of the most used types of software components: shared libraries. The paper builds on the idea of fault injection and how the fault injection process can be improved by analyzing library functions via symbolic execution.

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 and 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.

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. Library documentation can be incomplete and miss some of the error return codes. Last but not least, the documentation may not always be available or easy to parse. We must therefore extract information on the potential errors directly from the libraries. In this paper we introduce LibSE, a tool that can automatically infer possible return values by using symbolic execution.

Symbolic execution is a technique used among others on small-scale programs for automated testing, as it can find interesting program behaviors without any human assistance. For example, KLEE [2] found 56 serious bugs in some of the most extensively used and tested UNIX utilities. Instead of running the program with regular inputs, symbolic execution executes a program with symbolic inputs that are unconstrained (e.g., an integer input is represented as a symbol that can take on any integer value). When the program encounters a branch that depends on symbolic input, program state is forked to produce two parallel executions, one following the then-branch and another following the else-branch. The symbolic variables are constrained in the two clones with a predicate making the branch condition evaluate to true respectively false. Execution splits in two parallel sub-executions at each relevant branch. When an event of interest is encountered, solving the constraints collected along the path that led to the bug produces a set of inputs that reproduces it. In this way, symbolic execution can analyze

the behavior of code for entire classes of inputs, without having to try each one out, as in exhaustive testing. The biggest challenge in symbolic execution is path explosion: the number of execution paths to explore in a program grows exponentially in the number of branch instructions that depend (directly or indirectly) on inputs. As we'll argue, this problem is partially mitigated in our approach by the fact that we're not going to analyze entire programs but instead one library function at a time which empirically is likely to be small enough to be processed in a reasonable amount of time.

Another important concept we'll present is fault injection, a testing technique usually used to recreate corner-case situations (e.g., disk full, no memory available) which are easy to miss and can lead to crashes or correctness violations. More exactly, library-level fault injection involves selectivelly intercepting and failing library calls made by the system under test while observing its behavior. The specifics of library-level fault injection are beyond the scope of this paper but can be found in [5].

In the rest of the paper we provide an overview of LibSE (§2), describe the implementation (§3), show preliminary evaluation results (§5), survey related work (§6), and conclude (§7).

2. System Overview

The goal of LibSE is to give testers meaningful details about how a certain library function can fail in terms of return values vs. arguments, e.g. a possible output of our tools can be *the function can return NULL if the second argument is lower than 0 and the third is exactly 10*. We chose to determine the return values only by considering the function arguments as symbolic. While one may also choose to consider all global variables or perhaps the entire memory as symbolic, this can have adverse effects on the performance. We envision that future versions of LibSE will allow the user to specify what parts of the target's environment should be marked as symbolic and which should remain concrete.

In order to make the tool easily usable to users, we require no apriori knowledge of the library's internals. However, if the user has some domain knowledge (for example it knows that his application calls the function X with the second argument always set to 0) he could tune the LibSE results to his application, getting in return a potential speedup due to a smaller number of symbolic values.

It's not necessary to know the names of the exported functions as they can be determined along with their entry points either via the LLVM API or if the library is in native format, via tools like *readelf*.

Users point LibSE at a target library and they get in return, for each exported function, a list of pairs, each of them containing a function return value and a constraint (known as the path condition in symbolic execution vocabulary) that must be fulfilled when the function is called for the associated value to be returned. However, because the same value can possibly be returned via different execution paths, the final constraint will actually be a disjunction. This information can be either used by the developers in order to make sure his or her code properly uses the function and handles well the cases when the function fails, either by the testers in order to devise fault injection experiments as show in [5].

2.1. Alternatives

For our purposes, one alternative to symbolic execution is static analysis. Going from the idea that return codes are constant values, one could use a dataflow analysis algorithm to find the constants that can be propagated to the return value during the function execution [5]. While static analysis has the advantage of speed, its more likely to produce false positives when it deals with complex obfuscated code and it also has the disadvantage that it can't provide the constrains that symbolic execution does, e.g. one may end up thinking that a call to read can always fail with an EWOULDBLOCK error, while that is actually true only when the function receives an asynchronous file descriptor as argument.

More involved examples can be conceived and we found no general rule to handle them via static analasys.

3. Implementation

Our current implementation is based on the KLEE open source symbolic execution engine which runs on

the LLVM [4] framework. KLEE is a fast symbolic execution engine, reportedly scaling up to programs with 10,000 lines of code.

In order to use KLEE (and implicitly LibSE), one must first compile the code to LLVM bytecode. Fortunately, this is possible by using the C/C++ front-end LLVM provides. Although not all C/C++ code compiles via LLVM, software components as big as the FreeBSD kernel were compiled to LLVM and active development is underway.

To achieve our goal, we modified both the KLEE engine and the driver program that initializes and steers the symbolic execution. While the initial behavior was to start executing the target program with its main function, we added the ability to specify a custom function to start testing with. Using the LLVM API, we dynamically create code that calls uClibc for initialization and then passes control to our target function.

For the process to be as transparent as possible, we also dynamically instrument the function of interest in order to make all its arguments symbolic. While this could easily be done by modifying the source code before compilation and adding a `klee_make_symbolic` call for each variable that we want to consider as symbolic, we felt that it would not be reasonable to add this requirement. Therefore, our dynamic instrumentation enumerates through all function arguments, allocates a similar object in memory, calls the `klee_make_symbolic` function for the appropriate memory location and replaces all uses of the argument with the symbolic memory location. While this may seem overcomplicated, it is necessary because function arguments are considered by LLVM as 'registers', not as memory locations therefore the KLEE function responsible for marking objects as symbolic is unable to work with them.

One advantage of doing the analasys on LLVM's intermediate representation (IR) is the meta-data that is made available to us, for example the number of arguments a function expects and their types, information not available if running native code. A number of tricks or the user's input would be needed to achive the same result on native code. The same is true for global variables, which are clearly exposed via the LLVM API but are not straightforward to detect in x86 code.

We also modified the executor component of KLEE such that for each return instruction executed we check the stack depth at which we are. If we detect that the return instruction is in the frame of our target function we save the return value. The path condition are also saved by KLEE on disk both in its internal format and in CVC format.

4. Limitations

One of the limitations of our current implementation is that it relies on the LLVM representation of the program to be analyzed, which forces testers to recompile the libraries using the LLVM compiler. It would be therefore more natural to allow LibSE to work directly on the native binary format. This can be achieved in two ways: either dynamically translate the native code to LLVM as proposed by projects such as S2E [3] or use a different symbolic execution framework like BitBlaze [1] which can work directly on x86 code.

Furthermore, the LLVM toolchain is not 100% compatible with the gcc toolchain therefore recompiling an application is usually more involving that simply specifying a different compiler to the configure/makefile script.

Another aspect that limits the applicability of the tool for real software is that the current implementation doesn't support functions that accept complex type arguments. This is however only an engineering problem and will be addressed in future versions of LibSE.

Yet another limitation inherent to all tools based on symbolic execution is that it may not be possible to solve all constraints as this is a well known NP-complete problem. For example a test like

```
if (md5(arg1) == MAGIC)
```

is unlikely to be solved, therefore the paths leaving from this condition will usually not be explored leaving potential return values undiscovered. This happens because the paths are considered unfeasible (i.e. there is no input that can steer execution on that path). One could consider the approach where these paths are further explored optimistically, but this would have the big disadvantage of yielding false positives, not to mention the faster state explosion and the fact that the

symbolic execution engine won't be able to generate path conditions.

It is also possible to miss return values because of indirect branches, where the branch address is symbolic or based on a symbolic object (e.g. a virtual function call). In this case the symbolic execution engine is unlikely to infer a set of possible branch targets causing the abandon of the current execution path.

5. Evaluation

We evaluated LibSE on hand made toy libraries in order to verify its proper functioning and discovered as expected that the approach obtains accurate results that could otherwise be very hard to obtain via static analasys.

One such example is presented below:

```
int f(int x);
{
    int ret_val;

    if (0 == x) { return ERROR1; }
    if (x == x*2) ret_val = ERROR2;
    else ret_val = ERRORSUCCESS;

    while (x == 0x20 && (ret_val = 0x10))
    {
        ret_val = ERROR3;
        x = 0;
    }
    return ret_val;
}
```

Here the possible return values are ERROR1, ERRORSUCCESS and ERROR3. While these can be inferred via standard symbolic execution along with the constrains that the argument x must respect, it would arguably harder to devise a general static analasys algorithm to compute the same result.

6. Related Work

We built our system on top of the KLEE symbolic execution engine, initially designed for automatic software testing. Similar multi-purpose symbolic execution engines like BitBlaze have recently emerged and could be adapted for our goal.

Static analysis methods were used in LFI [5] in order to extract possible return values from arbitrary libraries in native (x86) format, achieving an average accuracy of 75%, however without offering argument constrains. The same paper explains how these results can be used to create fault injection experiments.

Previous work [6], 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. LibSE uses symbolic execution and is arguably providing more information that can help develop more accurate tests and thus eliminate false positives.

7. Conclusion

We presented LibSE, a tool for analyzing arbitrary library functions and finding possible return values along with the conditions needed to be satisfied for each value to be returned. We shown how we can augument an existing symbolic execution engine in order to achieve our goal and found promising initial results that suggest that the approach could scale and be usable on real libraries. We also explained how these results can be used to improve software development and testing via fault injection methods.

References

[1] D. Brumley, C. Hartwig, M. G. Kang, Z. L. J. Newsome, P. Poosankam, D. Song, and H. Yin. BitScope: Automatically dissecting malicious binaries. Technical report, Carnegie Mellon University, 2007.

[2] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th Symp. on Operating Systems Design and Implementation*, 2008.

[3] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *5th Workshop on Hot Topics in Dependable Systems*, 2009.

[4] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

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

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