Master's Thesis Project Description

Testing program resilience against deviant behavior

Patrick Emmisberger

June 27, 2016

1 Abstract

With the increasing demands on software quality, testing has become an important part of the software development process. In particular, *unit testing*[13][2] is a widely applied technique for detecting bugs in a modular and repeatable way. However, developing and keeping unit tests up-to-date is a time-consuming and tedious activity, that even when done meticulously, can still miss bugs. These bugs are often edge cases that are very hard to detect, because they are either counter-intuitive (otherwise the programmer would have tested for them) or hard to reconstruct (because they involve an unexpected behavior of an external resource like files, sockets, etc.)

On the other hand, *dynamic test generation* has been successfully applied in different contexts (in particular security vulnerabilities[1][15], API fuzzing[5] and regression testing[16]) to find these kinds of bugs. By modifying the input of a program randomly (black-box fuzzing) or analysing the program and deriving problematic inputs (white-box fuzzing[6] and test generation[16][4]), these edge cases can usually be found automatically, under the condition that the *unit-under-test (UUT)* does not exhaust the resources of the testing tool. However, most real-world applications are far too complex to be fully tested this way, as the time and memory consumption exceeds reasonable limits and as a result, bugs are missed (false negatives). Only applying dynamic test generation to methods lower in the call tree does not solve this problem, as it generates spurious errors (false positives).

In this project we try to address the limitations of both approaches, by building a platform that allows to selectively inject values (e.g. return values or arguments to a function call) during the execution of the UUT. This allows us to explore new paths in the code, without requiring the exhaustiveness of unconstrained dynamic symbolic execution and the associated resource problems. We do this directly on CPU instruction level, without the need for the source code of the UUT.

2 Core tasks

Unit tests typically cover frequently-exercised, non-failing paths, i.e. they ensure that given valid inputs, the program generates correct results. From the existing unit tests we get deep coverage (in terms of the call tree) directly, without any exploration. By only concentrating on certain inputs, we reduce the potential for resource exhaustion, but are still able to expand the execution from the successful path towards paths that may contain errors.

The central issue is the selection of the values that we want to manipulate. As a starting point, we will be focusing on the simulation of failing system calls. We inject artificial errors into system calls by returning an error, even if the system call was successful. This forces the unit test to deviate from the happy path and deal with different types of errors. This idea is particularly interesting, as it allows to test code, which typically would require the use of mocks or a complex test infrastructure (e.g. to simulate network errors). To prevent false positives, we need to only return errors that could potentially occur (i.e. malloc returning a null pointer is legal because running out of memory is always possible, while injecting an "Invalid file handle" is not, if the given handle is in fact valid). To accomplish that, we supply a configuration file to our value injection infrastructure, that encodes the fault model for a specific API (e.g. the set of functions used for file or socket IO) and a fault injection strategy. The fault model defines the interface that is used to communicate errors to the UUT and the logic for determining which faults to inject. The fault model is complemented with a fault injection strategy that defines the number and frequency of injected faults as well as a set of constraints that define the circumstances under which a fault will be injected.

This presents us with three main questions, which we seek to answer in this thesis. First, which are the APIs that promise the most potential for finding bugs? Second, given an API and a fault model, what are the strategies that find these bugs? And finally, which is the best fault injection strategy?

To implement the aforementioned features, it is necessary to instrument or fully replace the system calls that we want to support. Detours[8] allows us to rewrite functions in memory, to call a user-supplied replacement instead of the original function. Through a dynamically generated trampoline stub, the original function is still available and can be called as part of the replacement function. The infrastructure should be capable of handling different architectures (x86 and x64) and calling conventions. Support for concurrency is strictly required, as the intercepted system calls may run on different threads in parallel.

With the questions above in mind, we evaluate the success of different fault models and strategies based on the number of discovered bugs and increased code coverage minus any false positives introduced by our tool.

In summary, the core tasks are as follows:

1. Implement the instrumentation infrastructure.
2. Devise a flexible representation for the description of injection strategies and fault models.
3. Find and select APIs for fault injection and develop respective fault models.
4. Determine different injection strategies.
5. Evaluate the effectiveness of different injection strategies and API selections.

3 Extensions

As an extension, we apply our framework to different Microsoft products. Candidates are well-known desktop applications like Notepad, Internet Explorer or OneNote. This allows us to evaluate the framework on real-world applications with substantial test suites. In the later two cases, the applications contain a high degree of asynchronous network communication, which presents an interesting starting point for fault injection.

4 Related work

Most previous works on fault injection like the DOCTOR[7], FERRARI[9] and DEFINE[10] tools focus on the simulation of faulty hardware by manipulating either code or data. The aim is to check the robustness of software in the presence of arbitrary hardware failures. Our approach differs in the level where fault injection is applied (API vs. hardware) and also in the type of injected faults. Whereas the former tools simulate different types of memory corruption, we only inject faults that are permitted by the API specification (encoded as part of the fault model).

Jaca[12], a fault injection tool for Java, is also focused on interface error injection[3]. It corrupts parameter, field and return values to simulate bugs, and tests the robustness of the client in the presence of these corrupted values. Like with the former tools, the difference to our approach is that we only inject faults that are permitted by the API specification.

MAFALDA[14] and BALLISTA[11] use a similar instrumentation technique, but the aim is to test the robustness of the kernel for malformed system calls. These tools are the inverse to our framework in the sense that the role of producer and consumer is interchanged.

References

[1] Ella Bounimova, Patrice Godefroid, and David A. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *ICSE*, pages 122–131. ACM, 2013.

[2] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP*, volume 2374 of *LNCS*, pages 231–255. Springer, 2002.

[3] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa. Experimental analysis of binary-level software fault injection in complex software. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 162–172, May 2012.

[4] Pranav Garg, Franjo Ivančić, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *ICSE*, pages 132–141. ACM, 2013.

[5] Patrice Godefroid. Micro execution. In *ICSE*, pages 539–549. ACM, 2014.

[6] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151–166. The Internet Society, 2008.

[7] Seungjae Han, K. G. Shin, and H. A. Rosenberg. Doctor: an integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213, Apr 1995.

[8] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *Usenix Windows NT Symposium*, 1999.

[9] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: a flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, Feb 1995.

[10] Wei-Lun Kao and R. K. Iyer. Define: a distributed fault injection and monitoring environment. In *Fault-Tolerant Parallel and Distributed Systems, 1994., Proceedings of IEEE Workshop on*, pages 252–259, Jun 1994.

[11] P. Koopman and J. DeVale. The exception handling effectiveness of posix operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, Sep 2000.

[12] E. Martins, C. M. F. Rubira, and N. G. M. Leme. Jaca: a reflective fault injection tool based on patterns. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 483–487, 2002.

[13] Glennford J. Myers. *The Art of Software Testing*. Wiley, 1979.

[14] Manuel Rodríguez, Frédéric Salles, Jean-Charles Fabre, and Jean Arlat. *Dependable Computing — EDCC-3: Third European Dependable Computing Conference Prague, Czech Republic, September 15–17, 1999 Proceedings*, chapter MAFALDA: Microkernel Assessment by Fault Injection and Design Aid, pages 143–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[15] Alexander Sotirov. Windows animated cursor stack overflow vulnerability, 2007. `http://www.offensive-security.com/os101/ani.htm`.

[16] Nikolai Tillmann and Jonathan de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.