Available online at www.sciencedirect.com

SciVerse ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Systematic bug finding and fault localization enhanced with input data tracking

Jared D. DeMott[*], *Richard J. Enbody*[1], *William F. Punch*[1]

Department of Computer Science and Engineering, 3115 Engineering, Michigan State University, Lansing, MI 48824-1226, USA

ARTICLE INFO

Article history:
Received 21 February 2012
Received in revised form
18 July 2012
Accepted 17 September 2012

Keywords:
Testing and debugging
Software security
Fault localization
Distributed fuzzing
Information flow controls

ABSTRACT

Fault localization (FL) is the process of debugging erroneous code and directing analysts to the root cause of the bug. With this in mind, we have developed a distributed, end-to-end fuzzing and analysis system that starts with a binary, identifies bugs, and subsequently localizes the bug's root cause. Our system does not require the test subject's source code, nor do we require a test suite. Our work focuses on an important class of bugs, memory corruption errors, which usually have software security implications. Thus, our approach appeals to software attack researchers as well. In addition to our bug hunting and analysis framework, we have enhanced code-coverage based fault localization by incorporating input data tainting and tracking using a light-weight binary instrumentation technique. By capturing code coverage *and select input data usage*, our new FL algorithm is able to better localize faults, and therefore better assist analysts. We report the application of our approach on large, real-world applications (Firefox and VLC), as well as the classic Siemens benchmark and other test programs.

1. Introduction

A study conducted by NIST in 2002 reports that software bugs[2] cost the U.S. economy $59.5 billion annually (Tassey). More than a third of this cost could be avoided if better software testing was available. Debugging is part of this overall cost. Debugging is traditionally a labor-intensive activity where testers or developers seek to identify the erroneous code for which a given input is generating an error. Consequently, researchers seek techniques that automatically help locate the root of bugs.

Similar costs occur in the area of software exploitation. The tools and techniques to find and understand compiled-code mistakes are similar to testing and repair, except that once memory corruption bugs are understood they are weaponized into working software exploits. Countries such as China, Russian, and the U.S. are actively seeking to create cyber weapons (Clarke and Knake, 2010; Goel, 2011). The Stuxnet worm is a prime example of a cyber-weapon that leveraged memory corruption bugs in the Microsoft Windows operating system (Greengard, 2010). Therefore, cyber attackers also seek a system that can automatically find and analyze security-relevant bugs.

Given test cases (Voas and Miller, 1992), dynamic code coverage[3] information can be used to aid automatic, root-cause discovery of coding mistakes. This process is known

* *Corresponding author*. Tel.: +1 517 353 3541; fax: +1 517 432 1061.
E-mail addresses: jdemott@msu.edu, jdemott@vdalabs.com (J.D. DeMott), enbody@cse.msu.edu (R.J. Enbody), punch@cse.msu.edu (W.F. Punch).
[1] Tel.: +1 517 353 3541; fax: +1 517 432 1061.

[2] A software mistake or defect may be referred to throughout this document as an error, bug, or fault.

[3] Code coverage (CC) is a measure commonly used in software testing. It describes the degree to which the source code of a program has been tested. In this case, CC refers to the specific blocks that were executed for a given test.

0167-4048/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cose.2012.09.015

as *fault localization* (FL), and it is an active research area. Fault localization tools typically require (1) a test suite and (2) the software source code. A test suite will have both input and the ability to recognize output. The output of each test case is labeled, indicating whether that particular result is a "pass" or a "fail", where failure indicates a software-coding mistake. Our *fail* oracle is a program crash, caused by any type of memory corrupting bug capable of causing errors such as an access violation.[4] We chose to focus on this important subclass of bugs because cyber attackers often leverage memory corruption as the starting point for crafting software exploits to penetrate systems and networks. Our research therefore focuses on errors that tend to have software security implications. The advancements in our research are applicable for both defense (software reliability, debugging and repair) and offense (software attack, with the intent to exploit rather than repair). Debugging without source code is particularly relevant to offensive research, since these approaches will likely not have the source code they are analyzing.

We developed a distributed fuzzing system to identify, sort, and rate bugs. Having identified the existence of a fault, the next step is to localize the error within the code. A typical approach relies on the basic premise that code regions (often lines or basic blocks) that occur more frequently in failing traces[5] are more likely to contain the bug. These suspect regions are identified based on occurrence, and then ranked, creating a final list of suspicious locations. The more similar the passing and failing input test cases are, the better coverage-based algorithms tend to perform, because runtime differences are more easily identified.

We expand upon this general approach in two important ways:

1. We do not require a preexisting test suite to identify faults.
2. We do not require source code to perform fault localization.

These improvements are important because test suites may be incomplete or expensive to generate. In addition, source code may not always be available for legacy code or for offensive research. We address issue 1 by creating test suites automatically. First, we automatically download inputs by scouring the Internet. The quality of the inputs is judged based on code coverage. Second, we expand that set by systematically modifying the test inputs, a dynamic testing technique known as fuzzing (Miller et al., 1990). Issue 2 is addressed because fuzzing can operate on binaries where source is not available. Fuzzing automatically seeks memory corruptions, a critically important subclass of bugs that are the basis for most control-flow hijack vulnerabilities and exploits. Hackers use fuzzing to uncover bugs and create offensive tools. Commercial developers use fuzzing as a final test of their code—to keep ahead of the hackers.

Once fuzzing has identified an input that generates faulty behavior, we generate a smaller test suite (series of similar inputs) that is specific to the discovered bug, even though we don't yet know where the bug is located. As before, fault localization requires that the generated test suite have inputs that generate both passing and failing outputs. We can then apply a coverage-based algorithm to localize bugs to basic blocks—a unit of code that does not contain a branch.

Our work provides a number of contributions above and beyond those already mentioned. Our approach enhances results by increasing the suspiciousness score of blocks not thought to be noise (explained in Section 4.3.2). In addition, we increase accuracy through the addition of an input tainting, data-flow tracking algorithm (Section 4.3.3). For data flow tracking we tag key portions of the input data, and note the basic blocks within the program that operate on the tagged data. After our base FL algorithm scores each block, we then apply a score modifier (increased suspiciousness) to each basic block that uses tainted input data.[6] In this paper we will:

1. Describe our novel fault localization approach, and show that our algorithm outperforms previous approaches.
2. Describe our technique that allows us to work with closed-source code.
3. Detail the environment (system) in which our research currently operates.

1.1. Contributions

Our work differs from prior works in the following significant ways:

1. We are the first to create an end-to-end bug hunting and analysis framework that incorporates fuzzing, fault localization, and visualization. Fuzzing is used to find bugs. Input data analysis allows us to find the bytes that cause the fault and automatically create an FL test set. FL is used to identify the root cause of the bug. Visualization is used for repair or exploitation.
2. Our system can operate on closed-source applications, as well as programs where the source code is present.
3. We are the first to combine selective input tracking via data tainting with a traditional, code-coverage-based, fault localization approach. Since we know the bytes that caused the fault, only those bytes need be tracked, allowing our algorithm to key in on blocks that operate on suspicious input.

[4] Typical bugs of interest include stack buffer overflows, heap buffer overflows, off-by-one errors, integer errors, format string errors, incorrectly referenced memory, etc.

[5] A trace is a list of code regions encountered during one execution of the program given some input.

[6] Input data that is tracked throughout a programs lifespan is considered "tainted input". Tracking means the input will taint other data and registers as it is operated on in compiled code.