Infection Size as a Measure of Bug Severity

Mohammad R. Azadmanesh  
Università della Svizzera italiana, Lugano, Switzerland  
Mohammad.Reza.Azadmanesh@usi.ch

Matthias Hauswirth  
Università della Svizzera italiana, Lugano, Switzerland  
Matthias.Hauswirth@usi.ch

Abstract

A simple bug in a program can influence a large part of the program execution by spreading throughout the state at runtime. This is known as program infection. The seriousness of bugs is usually measured by studying their external effects. However, such effects essentially derive from internal factors of a program. Our idea is to focus on internal factors, in particular the infection chain, to measure how serious a bug was. This allows reasoning about bugs from a new and potentially insightful perspective.

Categories and Subject Descriptors D.2.8 [Software Engineering]: Metrics—BLAST

Keywords Bug severity, infection size, slicing, program trace

1. Introduction

We can find huge numbers of bugs in bug repositories, many of them resolved as fixed. One can rank them differently with respect to how serious they were. There are different metrics for doing that, such as the number of people affected by a bug, the costs of the consequences of the bug, the time required for fixing a bug, or the expertise of people needed in fixing a bug, just to mention a few.

We believe that the fixes in a bug repository provide valuable information about bugs that can be leveraged for further analysis about bugs for free. As a result, we approach this problem from another statistical perspective: What percentage of a program execution is infected by a bug? The higher the infection spread, the more serious the bug is.

Section 2 briefly describes the rationale behind our idea and Section 3 outlines a dynamic analysis approach to actually measure the infection chain size.

2. Rationale

We explain the rationale behind this idea with an analogy to the medical sciences. According to the Dorland’s Medical Dictionary for Health Consumers, an infection is the “invasion and multiplication of microorganisms in body tissues, especially that causing local cellular injury”. An infection spreads throughout the body, sometimes without any external sign, and suddenly reveals itself as the symptoms of an infectious disease. The more an infection spreads in the body, the harder it is to be treated. Besides, it can have side effects on the other parts of the body, if not treated in a timely manner.

An infection in a program starts with a bug, spreads throughout the state of a program at runtime, and finally shows up as a deviation from the expected observable behavior. The larger the affected output, the higher the possible costs for the users of the program. So, in general, measuring the infection size of a program can indirectly reflect the other criteria for measuring the seriousness of the bugs.

3. Approach

Figure 1 shows the timeline for two buggy program executions. Each row represents a step in the execution and each circle within a row represents a memory location involved in that step of the execution. The arrows represent information flow. In Figure 1(a), the bug occurs for the first time after running BC3, resulting in the infected state shown as a grey circle in the corresponding row. The infected state is used by another instruction (BC5), infecting its outcomes. This scenario is repeated up until a visible failure happens after running BC8. Figure 1(b) assumes the same execution history, but with a different event, instruction BC6, causing the infection.

Figure 1 shows that the spread in (a) is greater than in (b): There are 7 locations infected in (a), whereas there are only 2 infected locations in (b). That said, we base our measure of infection size on the number of memory location nodes infected by a bug.

In order to compute this measure, we use the concept of program slicing. Program slicing prunes a program such that those parts that could not have contributed to or derived from a slicing criterion are ruled out. A slicing criterion for a program specifies a window for observing its behavior.
Jackson and Rollins \cite{2} generalize the notion of program slicing to chopping: determining the subset of the program’s statements that cause influences of source elements on sink elements, where source is a set of definition ports and sink a set of use ports. We employ chopping to incarnate our idea. We perform the following three steps: (1) Finding the backward slice from the failure point, (2) Finding the forward slice from the bug point, (3) Counting the number of nodes existing in the intersection of these two slices.

**Step 1:** A backward slice prunes those parts that could not have contributed to a slicing criterion. Applying this notion to either part in Figure 1 with the slicing criterion set to the failure point, we end up with Figure 2 where the grey nodes and their generator instructions constitute the backward slice of the failure point.

**Step 2:** A forward slice prunes those parts that could not have derived from a slicing criterion. Applying this notion to Figure 1 with the slicing criterion set to the buggy point, we end up with Figure 3 where the grey nodes and their generator instructions constitute the forward slice of the buggy point.

As we said, we are going to take advantage of the knowledge already existing in bug repositories. As a result, we have access to both the buggy and the fixed version of the source code for each bug. In order to find the buggy point and the memory locations involved in that point of execution, we perform a diff on the source of the buggy version and the corresponding fixed version. Having found modified lines, we find those memory locations touched in these lines. These are the locations that are initially affected by the bug, so we mark them as the slicing criterion for this step.

**Step 3:** In this step, we just need to intersect the forward and the backward slice and count the resulting number of location nodes. For example, for the execution in Figure 1(a), the infection size is 7, whereas for the execution in Figure 1(b), the infection size is 2, i.e., the infection in (a) is larger, and thus potentially more severe, than the infection in (b).

4. Conclusions

We have implemented an early prototype of this approach on top of the BLAST dynamic analysis tool \cite{1} to check the feasibility of our idea for Java programs. We believe that understanding the infection sizes of bugs with existing fixes could help to better understand bugs and their severity. Moreover studying deeper characteristics of the infected part of the executions might lead to novel approaches for helping developers to debug and fix code.

Acknowledgments

The first author was funded by Swiss National Science Foundation grant 200021\_135245.

References

