Information and Software Technology 000 (2016) 1–15

Contents lists available at ScienceDirect

Information and Software Technology

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

Memory mutation testing

Fan Wu*, Jay Nanavati, Mark Harman, Yue Jia, Jens Krinke

CREST, University College London, WC1E 6BT, UK

ARTICLE INFO

Article history:
Received 15 August 2015
Revised 10 February 2016
Accepted 10 March 2016
Available online xxx

Keywords:
Mutation testing
Memory mutation

ABSTRACT

Context

Three decades of mutation testing development have given software testers a rich set of mutation operators, yet relatively few operators can target memory faults (as we demonstrate in this paper).

Objective

To address this shortcoming, we introduce Memory Mutation Testing, proposing 9 Memory Mutation Operators each of which targets common forms of memory fault. We compare Memory Mutation Operators with traditional Mutation Operators, while handling equivalent and duplicate mutants.

Method

We extend our previous workshop paper, which introduced Memory Mutation Testing, with a more extensive and precise analysis of 18 open source programs, including 2 large real-world programs, all of which come with well-designed unit test suites. Specifically, our empirical study makes use of recent results on Trivial Compiler Equivalence (TCE) to identify both equivalent and duplicate mutants. Though the literature on mutation testing has previously deployed various techniques to cater for equivalent mutants, no previous study has catered for duplicate mutants.

Results

Catering for such extraneous mutants improves the precision with which claims about mutation scores can be interpreted. We also report the results of a new empirical study that compares Memory Mutation Testing with traditional Mutation Testing, providing evidence to support the claim that traditional mutation testing inadequately captures memory faults; 2% of the memory mutants are TCE-duplicates of traditional mutants and average test suite effectiveness drops by 44% when the target shifts from traditional mutants to memory mutants.

Conclusions

Introducing Memory Mutation Operators will cost only a small portion of the overall testing effort, yet generate higher quality mutants compared with traditional operators. Moreover, TCE technique does not only help with reducing testing effort, but also improves the precision of assessment on test quality, therefore should be considered in other Mutation Testing studies.

1. Introduction

Mutation testing is an effective fault-based testing technique that aims to identify whether a codebase is vulnerable to specific classes of faults [1]. In mutation testing faults are deliberately seeded into the original program, by simple syntactic changes, to create a set of faulty programs called mutants, each containing a different syntactic change. By carefully choosing the location within the program and the types of faults, it is possible to detect vulnerabilities that are missed by traditional testing techniques [2,3], to simulate any test adequacy criteria [4] whilst providing improved fault detection [5,6].

Memory errors are one of the oldest classes of software vulnerabilities that can be maliciously exploited [7]. Despite more than two decades of research on memory safety, memory vulnerabilities have been still ranked in the top 3 of the CWE SANS top 25 most dangerous programming errors [8]. Recent work on memory

* Corresponding author. Tel.: +44 (0)2076793058.
E-mail addresses: fan.wu@ucl.ac.uk (F. Wu), jaysnanavati@hotmail.co.uk (J. Nanavati), mark.harman@ucl.ac.uk (M. Harman), yue.jia@ucl.ac.uk (Y. Jia), jens.krinke@ucl.ac.uk (J. Krinke).

http://dx.doi.org/10.1016/j.infsof.2016.03.002
0950-5849/

vulnerability detection [9–11] in C applications has shown the existence of a wide range of vulnerabilities such as uninitialised memory access, buffer overruns, invalid pointer access, beyond stack access, free memory access and memory leaks in published code. Moreover, these vulnerabilities are highly prone to exploitation. For example, vulnerabilities such as buffer overflows when using `malloc()` facilitate exploits that overwrite heap meta-data, gain access to unavailable function/data pointers, overwrite arbitrary memory locations, and create fake chunks of memory that may contain modified pointers.

Traditional mutation operators only simulate simple syntactic errors based on the Competent Programmer Hypothesis [12]. Mutants generated using these operators may drive testers to generate test suites mainly targeting simple syntactic errors. Semantic mutation operators on the other hand seek to mutate the semantics of the language [13]. The semantic mutants can capture the possible misunderstandings of the description language and thus capture the class of semantic faults. However, both traditional and semantic mutation operators are not designed to find test cases revealing memory faults, thereby creating a weakness in traditional Mutation Testing.

To mitigate this limitation, there has been an attempt to design mutation operators for a specific type of memory vulnerabilities, Buffer Overflow vulnerabilities [14]. This work proposed 12 mutation operators that seek to simulate Buffer Overflow by making changes to the related vulnerable library functions and program statements. However, the proposed operators do not consider other general memory vulnerabilities, such as uninitialised memory access, `NULL` pointer dereferencing nor memory leaks caused by faulty heap management.

To address this problem we design 9 Memory Mutation Operators, simulating three classes of common memory faults. We also introduce two additional weak killing criteria, i.e. Memory Fault Detection and Control Flow Deviation for memory mutants. Because memory faults do not necessarily propagate to the output, making the strong killing criterion, which is widely adopted in traditional Mutation Testing, inadequate to detect such faults. A single Mutation Testing tool was developed using both of the traditional and Memory Mutation Operators with the traditional strong killing criterion and the proposed weak killing criteria also incorporated.

We compare the effectiveness of Memory Mutation Operators against traditional mutation operators using 18 subject programs with a variety of sizes. Our results show that our memory mutants introduced memory faults that cannot be simulated by traditional mutation operators. We also study the difference between traditional strongly killing criterion with the proposed weakly killing criteria. The results show that, among 1536 generated memory mutants (with 90 TCE-equivalent or duplicate mutants excluded), traditional strong killing criterion killed only 43% of the mutants, leaving 869 mutants unkilled. We also find the two new memory killing criteria introduced are more effective at distinguishing memory mutants, killing up to 80% of those survived mutants across all subject programs.

This paper is an extension of our work published at Mutation'15 [15]. The differences between this extended work and the previous work include the following: 1) we extended our memory mutation tool to support traditional selective mutation, and refined the weakly killing criteria according to the feedback of the previous work; 2) using the extended tool, we added new experiments to compare the prevalence and quality of traditional mutants and memory mutants; to make our evaluation more thorough, we reran all the experiments with additional analysis to reveal some equivalent mutants and duplicated mutants using the recently-introduced TCE equivalence technique [16]; 3) furthermore, we added two additional large subjects and ran all the experiments on them as a case study. The quantitative results are different from the previous work due to more precise analysis, yet the conclusions remain consistent with the previous work. The primary contributions of this paper are as follows:

1. The design of 9 Memory Mutation Operators to mimic several categories of memory faults. The mutants generated form these operators can be used to select tests that mitigate memory vulnerabilities.
2. A comprehensive empirical study exploring the characteristics of memory mutation operators and a further empirical study to compare them with the traditional operators. On 16 subject programs, Memory Mutation Operators successfully insert memory faults and generate 368 mutants, 94% of them cannot be simulated by traditional mutation operators. A case study using 2 large programs demonstrates that Memory Mutation Operators are feasible to scale to large programs.
3. The introduction of Memory Fault Detection (using *Valgrind* for precise assessment of memory faults) and Control Flow Deviation as additional killing criteria. This is the first weak killing criteria proposed for memory mutation and the results show that up to 80% of surviving mutants are killed by these additional criteria.
4. An open source C mutation testing tool [1] that features both traditional and Memory Mutation Operators. The tool also supports traditional strong killing criteria as well as the Memory Fault Detection and Control Flow Deviation killing criteria.

The rest of the paper is organised as follows: background theory and the problem statement are presented in Section 2, while the methodology including Memory Mutation Operators and proposed new killing criteria are presented in Section 3 together with a list of research questions. Section 4 introduces the Mutation Testing framework and experimental setting, the results and analysis of which are shown in Section 5. We summarise the threats to validity and related work in Sections 6 and 7 respectively, followed by conclusions in Section 8.

2. Background

Mutation Testing [1] is a white box testing technique that measures the quality/adequacy of tests by examining whether the test set (test input data) used in testing can reveal certain types of faults. A mutation system defines a set of rules (mutation operators) that generate simple syntactic alterations (mutants) of the Program Under Test (PUT), representing errors that a "competent programmer" would make, known as the Competent Programmer Hypothesis (CPH) [12].

To assess the quality of a given test suite, the set of generated mutants are executed against the input test suite to determine whether the injected faults can be detected. If a test suite can identify a mutant from the PUT (i.e. produce different execution results), the mutant is said to be *killed*. Otherwise, the mutant is said to have *survived* (or to be live). A mutant may remain live because either it is equivalent to the original program (i.e. it is functionally identical to the original program although syntactically different) or the test suite is inadequate to kill the mutant. The Mutation Score (MS) is used to quantify how adequate a test suite is in detecting the artificial faults. It is calculated as the following formula:

$$MS(P,T) = \frac{\text{number of mutants killed}}{\text{total number of non-equivalent mutants generated}}.$$

P is the program under test and T is the set of tests. However, it is very hard and generally undecidable to determine the exact

[1] https://github.com/jaysnanavati/Mutate