INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING AND APPLICATIONS
VOLUME XVIII, ISSUE V, May 2024, WWW.IJCEA.COM, ISSN 2321-3469

HAVE WE GOTTEN THERE? MUTATION ANALYSIS AS AN AUTOMATED PROGRAM ANALYSIS TECHNIQUE FOR LINUX KERNEL'S REGRESSION TEST SUITES GENERATION

Xaveria Youh Djam[1], Nachamada Vachaku Blamah[2]

[1]*Department of Computer Science, University of Yaounde I, PMB 812 Yaounde- Cameroon.*

[2]*Department of Computer Science, University of Jos, PMB 2084 Jos -NIgeria.*

ABSTRACT:

Mutation testing is a fault-based program analysis technique to assess and improve the quality of a test suite. However, it suffers from two main issues. First, there is a high computational cost in executing the test suite against a potentially large pool of generated mutants. Second, there is much effort involved in filtering out equivalent mutants, which are syntactically different but semantically identical to the original program. Despite three decades of research on this technique, prior work has mainly focused on detecting equivalent mutants after the mutation generation phase, which is computationally expensive with limited efficiency and its usage in large systems is still rare. In this paper, we propose a novel two-stage mutation testing framework for bug-proneness as a criterion to measure the effectiveness of test suite. This framework offers a method to tackle the complexity of large programs by reducing the number of mutants used. We present our experimentation using mutation analysis on Linux Kernel's regression test suites generation on RCU (Read Copy Update) module, where we adapt existing techniques to constrain the complexity and computation requirements. We show that mutation analysis can be a useful tool, uncovering gaps in even well-tested modules like RCU. This experiment has so far led to the identification of 2 gaps in the RCU test harness, and 2 bugs in the RCU module masked by those gaps. We argue that mutation testing can and should be more extensively used in practice.

Keywords: Software Testing, Mutation Testing, Linux Kernel, Regression Testing.

[1] Introduction

Software testing is an indispensable, error prone, tedious and most expensive part of the software development life cycle. Moreover, unprecedented increased of software usage in our daily lives reinforced the need for quality testing. Furthermore, the exponential growth of software system and the demand for reusable software components, automating the software testing processes is highly desirable. Unfortunately, the inherent complexity of modern software makes the adequacy of evaluating quality difficult. Failure to meet quality requirements can lead to tremendous cost, especially when these lead to software failures [1]. It is not impossible for a test case to account for all possible scenarios that can arise during the execution of a program. Thus, defect free software is an illusion and almost all programs contain defects that lead to more or less severe failures. Nevertheless, after the initial development, like any software engineering artifact, source code requires quality assessment and maintenance. The essence of software testing is discover hidden faults in the System Under Test (SUT) but unfortunately, the inadequacy of existing techniques in dealing with fault detection capability necessitated this research. To this end, one is interested in a program analysis technique for test suits generation and test coverage.

Mutation testing is a fault-based and program analysis testing technique which provides a testing criterion that can be used to measure the effectiveness of a test suit in terms of the ability to detect faults [2]. According to a recent work by Djam et al. [3], a comparative analysis of Prime-Path Coverage (PPC), Edge-Pair Coverage (EPC) and Edge Coverage (EC) based on mutation analysis criterion was conduction. The experimental results show that mutation analysis criterion is effective fault detection.

Despite the extensive used of mutation analysis in literature, there are some painful drawback inherent with mutation analysis for large and complex problems. Mutation analysis is rarely used in the industry because of the large numbers of mutants generated for huge and complex real-world projects, which must be analysed, thus computational

cost is highly increased. Another reason is the lack of proof of mutation analysis to effectively handle equivalent mutants and human oracle problems.

Exhaustive mutation analysis generates large numbers of mutants, but mutant sampling [3] and mutant execution optimizations [4,5] can help to mitigate the problem. Offutt et al. [6] categorized these efforts into do fewer, do faster, and do smarter approaches. Unlike model checking, mutation analysis doesn't require any kind of modeling of the environment and associated data structures, which makes it more easily applicable to complex systems. Mutation analysis has been widely adopted by academia than industry and the technique is mostly using relatively simple program.

Though testing is effective in ensuring software quality, as software systems get more complex, the task of exhaustive testing becomes more complex and even infeasible in some cases. In order to build less error prone systems, we, therefore, need to not only focus on quickly and efficiently identifying bugs through testing and verification of software, but also on identifying factors associated with bugs that could be used in fault prediction techniques to potentially help us focus quality-assurance efforts on the most defect-prone parts of the code. The quality of software artifacts is one of the key concerns for software practitioners and is typically measured through effective testing. While it is widely held that "you cannot test quality into a product," you can use testing to detect that the Software Under Test (SUT) has a fault, and to estimate its likely overall quality. Moreover, while testing itself does not produce quality, it leads to the discovery of faults. When these faults are corrected, software quality improves.

Automated testing is now a mainstream not only to software industry but equally to the research community. The Linux Kernel is one of today's most complex and fast evolving software that maintaining quality assessment is pretty difficult but must not be avoided. Applying techniques such as program analysis and model checking on a well tested software artifacts like kernel and its modules to uncover gaps is a daunting task due to

the size and complexity of the source code. Although mutation analysis can be applied on Linux Kernel, it is not trivial to do so. As this research area matures, it is essential to conduct mutation analysis on complex programs in order to provide many benefits to the broader community of researchers and practitioners. This paper describes our experience using mutation testing on the Linux-kernel's RCU to perform regression testing.

The main contributions of this article are three-fold:

1. We introduced a novel two-stage mutation testing framework for bug-proneness as a criterion to measure the effectiveness of test suite. This framework offers a method to tackle the complexity of large programs by reducing the number of mutants used.

2. Empirical evaluation of effectiveness of test suite quality measurements such as statement coverage and mutation score and identifying mutation score as a better criterion between the two.

3. Investigating the applicability of mutation analysis in real-world complex software system adapting existing techniques and showing its effectiveness.

The remainder of study is organized as follows: Section 2 discusses background and related work. Section 3 describes our research method, including the overall process. Section 4 presents the results of applying mutation analysis on a complex and well-tested software and hence discusses the potential threats to validity of our study. Finally, Section 5 concludes this study and states the future work directions.

[2] Background

The view of mutation analysis as a process of seeding faults into a SUT is established firmly in the literature [6,8]. Mutation testing has long been used to compare test sets and criteria by using mutants as proxies for faults [8,9].

[2.1] Mutation Analysis

The idea of mutation analysis was first proposed by Lipton [10]. Mutation analysis seeks to evaluate test suites by embedding known defects into the SUT. These defects are called mutants, and are produced by simple syntactic rules, e.g., changing a relational operator from “>” to “>=”. These rules are called mutation operators. The term mutagen is synonymous with both mutation-operator and mutation transformer in literature. If only one single mutation separates the mutant from the original program, it is called a first order mutant (FOM). A higher order mutant (HOM) is separated from the original by multiple mutations.

[2.2] Read-Copy Update in the Linux Kernel

The Read-Copy Update (RCU) is a scalable, high performance Linux-kernel synchronization mechanism that runs low-overhead readers concurrently with updaters [11,12]. Producing quality RCU implementations are decidedly non-trivial and their stringent validation is mandatory. Over the past 25 years, many technologies have been added to the Linux kernel, one example being Read-Copy Update (RCU) [13]. Because RCU is used on large clusters and has been extensively tested, most remaining bugs are likely to be in difficult-to-reach parts of the code. RCU is used in read-mostly situations. Readers run concurrently with updaters, so RCU maintains multiple versions of objects and ensures they are not freed until all pre-existing readers complete, after a grace period elapses. The idea is to split updates into removal and reclamation phases [14]. The removal phase makes objects inaccessible to readers, waits during the grace period, and

then reclaims them. Grace periods need wait only for readers whose run time overlaps the removal phase. Readers starting after the removal phase ends cannot hold references to any removed objects and thus cannot be disrupted by objects being freed during the reclamation phase. Modern CPUs guarantee that writes to single aligned pointers are atomic, so readers see either the old or new version of a data structure. This enables atomic insertions, deletions, and replacements in a linked structure. Readers can then avoid expensive atomic operations, memory barriers, and cache misses. In the most aggressive configurations of Linux-kernel RCU, readers use the same sequence of instructions that would be used in a single-threaded implementation, providing RCU readers with excellent performance and scalability [15].

The bulk of RCU is in 4 files (srcu.c, tiny.c, update.c and tree.c). Together, these only total to 5,542 lines of code (LOC), with the largest being 3,771 LOC. The RCU is therefore not the largest program examined using mutation analysis (Apache Commons Math, with 202,000 lines of code, was analyzed by Gopinath et al. [16]), but it has the highest complexity, as Apache commons is a large but shallow set of library calls. RCU's primary test system, rcutorture, is an automated stress-testing mechanism composed of 1,800 lines of code. rcutorture can simulate 12 different RCU scheduling variations and test on 16 hardware configurations. These configurations are specified using parameters such as CONFIG_NR_CPUS, CONFIG_HOTPLUG_CPU, CONFIG_SMP, etc. rcutorture uses Qemu to load kernels built using these parameters and monitors their performance for a user specified period. The test periodically outputs status messages via printk(), which can be examined via the dmesg command. Qemu uses KVM, essentially running a virtualizer (Qemu) on top of another virtual machine, a practice referred to as

nested virtualization [17]. Interest in rcutorture has grown, with the number of emerging contributions growing between 2010 and 2020

[3] Mutation Analysis as an Automated Program Analysis Technique for Linux Kernel Test Suites Generation

Our proposed mutation testing framework consists of the following two stages:

(1) Mutation-based Test Code Engineering

(2) RCU test code Engineering

[3.1] Mutation-Based Test Code Engineering

Our mutation-based test code engineering consist of designing program mutation operators and mutation testing metric for Linux Kernel that assist in investigating the applicability of mutation analysis in real-world complex software system

[3.1.1] Program Mutation Operators

The ability of mutation analysis to detect faults is based on the mutation operator used. In order to generate mutant for the subjects programs,

Table 3.1: Mutation operators

Operators Name	Description
rep_int_const	Replace any integer constant C by 0, 1, -1, (C+1) or (C-1)
rep_op	Replace an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class
Neg_op	Negate the decision in an “if”, “while” statement or any control structure
del_stmt	Delete a statement
Omit_meth_call	Omit method call Suppress a call to a method. If the method has a return value, a default value is used instead.

The first three operator classes were extracted and modified from Offutt et al.'s research [24] on identifying a set of "sufficient" mutation operators, i.e., a set S of operators such that test suites that kill mutants formed by S tend to kill mutants formed by a very broad class of operators. They were adapted so that they would work on C programs rather than the Fortran of the original research. The fourth and firth operators, was added to handle

pointer-manipulation and field-assignment statements that would not be vulnerable to any of the sufficient mutation operators.

Table 3.2 contains some sample mutants from RCU and Table 3.3 contains the details of mutants for each mutation operator category.

Table 3.3: Mutation operators

Operators Name	Mutant Versions				
	Tree.c	Tiny.c	Update.c	Srcu.c	Total mutants
rep_int_const	882	234	56	34	1206
rep_op	545	35	45	56	681
Neg_op	434	456	145	45	1080
del_stmt	623	124	45	245	1037
Omit_meth_call	335	345	34	123	837
Total mutants	2891	1194	325	503	4841

[3.1.2] Mutation Testing Metric for Linux Kernel

Suppose we have a given program P under analysis, mutation testing applies a set of mutation operators to generate a set of mutants M for P. Each mutation operator applies transformation rules such as: negating a conditional statement from if $(x>0)$ to if $(x<=0))$ to generate mutants; replacing an integer by certain criteria, replacing an arithmetic operator by another operator in the same class, deleting a program statement, or omitting method calls. Each mutant $m \in M$ is the same as the original P except the mutated program statement. Then, all the mutants in M are executed against the test suite T of P to evaluate

its effectiveness – for each mutant *m*, when the execution of $t \in T$ on m has different result from the execution of *t* on *P, m* is killed by *t*; otherwise, *m* survives. In this way, mutation testing results can be represented as a mutation matrix (Table 3.3).

[3.2] RCU Test Code Engineering

After applying the mutation generator to each of RCU's files in Table 3.3 (less than 10 minutes for all files), the next step was to compile the 4,841 resulting mutated versions of RCU. For scalability reasons, load and stress testing were conducted on four virtual machines running in parallel built on the ESXi 5.6 platform [20]. After compilation, we had to test each of the mutants. Running this testing serially would take excessive amounts of time. The kernel cannot run as a thread, so we could not use threads to parallelize the testing. The logical step was therefore to use four virtual machines in parallel.

The next step was to run the mutated RCU's to determine if rcutorture would flag them. Because execution and detection of faults is probabilistic, we allocated relatively short timeouts (2 minutes). We hypothesized that most faults would be trivially detected, while a handful of faults require very long runtimes. Our goal was to narrow the set as quickly as possible to then allocate more time and resources to the hard mutants. Each virtual machine was assigned to handle one specific mutant. rcutorture uses Qemu to load different versions of the kernel, built using permutations of a set of parameters On each virtual machine, 14 parallel processes were set up to compile 14 different kernel images using these parameters. This helped us to cut the setup time down by 1/14. Next, a single sequential process would load the images on Qemu and monitor the thread for 2 minutes. We used a single process because all Qemu processes were killed after 2 minutes, which

would kill all instances of Qemu. If we had run 14 Qemu instances in parallel, all would be killed when the first finished.

[4] RESULTS AND DISCUSSION

We now discuss our experiments, which were performed on a 64-bit machine running on Linux Mint 20.2 with Intel Corei7, 3.07 GHz processing speed with 16 GB of memory. The source code of our RCU model and the experimental data are available at https://github.com/testing/verify-mutantrcu/releases/tag/ date23-camera-ready.

[4.1] RESULTING PATCHES TO RCU

In this section we list the patches that resulted from our application of mutation analysis on RCU along with a brief description. we consider patches whose commits are explicitly linked to a bug report from the Linux Kernel Bugzilla tracking system. All patches were constructed via rcutorture process P, hereafter referred to as P-Patches.

P-Patch 1: Test both RCU-sched and RCU-bh for Tiny RCU Tiny RCU provides both RCU-sched and RCU-bh configurations, but only RCU-sched was tested by the rcutorture previously. This gap was identified via mutation analysis on tiny.c. This commit changed the TINY02 configuration to test RCU-bh, with TINY01 continuing to test RCU-sched.

P-Patch 2: Correctly handle non-empty Tiny RCU callback list with none ready This fixes an RCU bug. This bug is most likely to occur if there is a new callback between the time rcu_sched_qs() or rcu_bh_qs() is called before __rcu_process_callbacks() is invoked. This bug was detected by the addition of RCU-bh to rcutorture.

P-Patch 3: Test SRCU cleanup code path: An rcutorture memory leak of the dynamically allocated >per_cpu_ref per-CPU variables was identified via our mutation analysis. This commit adds a second form of srcu (called srcud) that dynamically allocates and frees the associated per-CPU variables. This commit also adds a cleanup() member to rcu_torture_ops that is invoked at the end of the test, after - >cb_barriers(). After the

patch, the SRCU-P torture-test configuration selects scrud instead of srcu, with SRCU-N continuing to use srcu, thereby testing both static and dynamic srcu_struct structure.

[4.2] DISCUSSION

Sequel to the above process, we were able to narrow 4,348 mutants to only 527 potentially interesting mutants with little or no human intervention. While 527 may seem like a large number, it is very likely that this could be further reduced by giving rcutorture more run-time to try to kill these mutants. We look at our process as a kind of mutation analysis pre-processing, where we, as quickly as possible, with maximum automation, narrow the field of mutants to the set of interesting mutants.

Given the complexity of RCU, one could expect to see most mutants fail during compilation. However, only 13% of generated mutants failed to build. Most of these failing mutants came as a result of mutating function or other parameters in a way that causes a conflict, which the compiler will catch. This is an indication that the mutation framework is doing a reasonably good job of only creating plausible mutants rather than randomly changing tokens in the code. For a simpler application, we would expect to see an even lower failure rate

We found that about 8% of our mutants were equivalent mutants, which is close to the findings of Kintis et al. [18]. When we look at unique mutants in each file we see that

tree.c has the highest percent of unique mutants (82%). This is the biggest file, with 101 functions. tree.c implements a large part of RCU's synchronization.

[4.3] THREADS TO VALIDITY

We focus on a subset of Linux Kernel and we used the tool by Andrews et al. [19] to generate mutants. Using different mutation operators or tools could lead to different results. Our study looked at a program written in C, so additional studies on large projects in other programming languages would be needed to verify the same benefits there.

[5] CONCLUSION AND FUTURE WORK

The biggest challenge in the software development industry is to deliver an application with 100% defects free. This paper describes the applicability of mutation analysis in a real-world complex and well test software system as Linux Kernel. We found that mutation analysis can uncover interesting instances of weak testing, even in a robust system like rcutorture. This work shows that RCU is a rich example to drive research: it is small enough to provide models that can just barely be verified by existing tools, but it also has enough concurrency and complexity to drive advances in techniques and tooling. While a fairly large number of mutants were left alive after our initial run, subsequent runs should further reduce the surviving mutants.

In order to further improve on our work, we plan to implement new mutation operators in order to make a comparative evaluation with data coverage techniques and symbolic execution. In addition, because rcutorture and kernel testing is a non-deterministic process, it is likely the case that a set of short runs is more efficient for killing mutants

than longer runs. We will investigate this in our future work by integrating search-based optimization techniques such as genetic algorithm and hill climbing.

Acknowledgments

We thank the anonymous reviewers for the valuable comments. This work is supported in part by AFRICOM Project Grant No. AFR21122.

References

[1] Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing, TSE, vol. 37, no. 5, pp. 649–678.

[2] Ammann, P. and Offutt,J. (2016). Introduction to software testing. Cambridge University Press.

[3] Djam, X.Y., Blamah, N.V. and Ezema M.E(2021). A Comparative Evaluation of Test Coverage Techniques Effectiveness. Journal of Software Engineering and Applications, 14, vol. 14, No. 6, 95-109.

[4] Adamopoulos, K. Harman, M. and Hierons. R.M. (2004) How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation using Co-Evolution. In Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO '04), volume.

[5] Zhang, L., Hou, S. S., Hu, J. J., Xie, T., & Mei, H. (2010). "Is operator-based mutant selection superior to random mutant selection?". In Software Engineering, 2010 ACM/IEEE 32nd International Conference on (Vol. 1, pp. 435-444). IEEE.

[6] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., & Zapf, C. (1996). "An experimental determination of sufficient mutant operators". In Transactions on Software Engineering and Methodology, Vol. 5, No. 2, (pp. 99-118).

[7] B. Kurtz, P. Ammann, M.E. Delamaro, J. Offutt, L. Deng, Mutant subsumption graphs, in: Tenth IEEE Workshop on Mutation Analysis, Mutation 2014.

[8] Just, R. and Schweiggert, F, (2015). Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators, Softw. Test. Verif. Reliab. Vol. 25, 490–507.

[9] Namin, A.S., Andrews, J.H. and Murdoch, D.J. (2008). Sufficient mutation operators for measuring test effectiveness. In Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pages 351–360.

[10] Lipton, R. (1971). "Fault diagnosis of computer programs". Student Report, Carnegie Mellon University.

[11] McKenney, P. E. (2013). "Structured deferral: synchronization via procrastination". In Communications of the ACM, Vol. 56, No. 7, (pp. 40-49).

[12] Guniguntala, D., McKenney, P. E., Triplett, J., & Walpole, J. (2008). "The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux". IBM Systems Journal, Vol. 47, No. 2, (pp. 221-236).

[13] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in PDCS, 1998.

[14] P. E. McKenney and J. Walpole, "Introducing technology into the Linux kernel: a case study," ACM OSR, vol. 42, no. 5, 2008.

[15] J. Tassarotti, D. Dreyer, and V. Vafeiadis, "Verifying read-copy-update in a logic for weak memory," in PLDI, 2015.

[16] Gopinath, R., MA Alipour, Ahmed, I., Jensen, C., & Groce, A. (2016). "On The Limits of Mutation Reduction Strategies". In Proceedings of the 38th International Conference on Software Engineering, (pp-511-522). ACM.

[17] McKenney, P. E., Eggemann, D., & Randhawa, R. (2013). "Improving energy efficiency on asymmetric multiprocessing systems".

[18] Kintis, M., Papadakis, M., & Malevris, N. (2010). "Evaluating mutation testing alternatives: A collateral experiment". In Software Engineering Conference (APSEC), 2010 17th Asia Pacific, (pp. 300-309). IEEE.

[19] Andrews, J.H., Briand, L.C. and Labiche. Y. (2005). Is mutation an appropriate tool for testing experiments? In Proceedings of the 27th International Conference on Software Engineering,

ICSE '05, pages 402–411.

[20] ESXI: http://searchvmware.techtarget.com/definition/VMware-ESXi