Applying Mutation Analysis On Kernel Test Suites: An Experience Report

Iftekhar Ahmed, Rahul Gopinath,
Carlos Jensen
School of EECS
Oregon State University, Corvallis, USA
{ahmedi, gopinath, cjensen}@eecs.orst.edu

Alex Groce
School of ICCS
Northern Arizona University, Flagstaff, USA
agroce@gmail.com

Paul E. McKenney
IBM Linux Technology Center
paulmck@linux.vnet.ibm.com

Abstract— **Mutation analysis is an established technique for measuring the completeness and quality of a test suite. Despite four decades of research on this technique, its use in large systems is still rare, in part due to computational requirements and high numbers of false positives. We present our experiences using mutation analysis on the Linux kernel's 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 3 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—Mutation Analysis; Linux kernel

I. Introduction

Quality is important for software systems. Unfortunately evaluating quality, especially the presence of bugs, becomes more difficult as the complexity of that software increases. The best way to ensure software meets quality requirements is to engage in extensive and thorough testing. The goal of testing is to discover faults in the System Under Test (SUT) by executing tests that create conditions that lead to failure, and detect these.

Two important "problems" with testing are knowing when you have performed sufficient testing and determining whether your testing is biased. A well-respected technique for evaluating test suites and test coverage is mutation analysis. Unlike code coverage and other benchmarking techniques, mutation analysis addresses the oracle problem as well as determining the degree to which SUT behaviors are explored. By inserting random but realistic bugs [6, 10, 22] in the SUT, known as mutants, we can determine a tests ability to uncover faults, not just its ability to explore behavior. The ratio of mutants found over all mutants, is used as the test suite's effectiveness (mutation score). Mutation analysis thus identifies gaps in the test suites and subsumes almost every other test adequacy criteria [7, 13, 37, 44].

One of the primary reasons mutation analysis is rarely used on large and complex programs is that mutation analysis generates large numbers of mutants, which must be analyzed, though mutant sampling [40, 45, 47] and mutant execution optimizations [21, 41, 48] can help to mitigate the problem. Another reason is the lack of proof of mutation analysis' applicability to complex real world projects. Thus, mutation analysis has been more widely adopted by academia than industry, and the technique mostly evaluated using relatively simple programs and test suites.

The Linux kernel is one of today's more complex software systems, evolving so rapidly that maintaining quality assurance is hard [12, 39]. Applying techniques such as code analysis and model checking on the kernel and its modules is difficult because of their size and the complexity of the code. Source code analysis generates a large number of false positives and warnings [20] which need to be screened by domain experts, which is expensive in terms of both time and resources.

Although mutation analysis can be applied to the kernel, it is not trivial to do so. First, one has to generate an enormous set of mutants. Second, one needs to compile and run each mutated version of the code and put it through the test harness. As with many complex systems, execution is probabilistic, meaning that the amount of time needed to "kill" a mutant (detecting it with the test suite) cannot be determined a priori. Not finding a fault at time *j* could be due to inefficient tests, or insufficient run time. Therefore, to avoid false positives, a test suite would need infinite run-time on each mutant. This is clearly not feasible, and a probabilistic approach must be adopted.

This paper describes our experience using mutation testing on the Linux-kernel's RCU [43]. Our goal was to determine whether: (1) mutation testing RCU is feasible, and (2) whether it can uncover bugs in RCU. Locating bugs in RCU is hard because RCU is well tested and heavily used: About one in 2,000 lines of kernel code uses RCU [32], and it has been a favorite target for model checkers [2, 15, 28, 29]. If mutation testing can locate new bugs, these are likely deep, and the technique can be useful for locating bugs in other complex software.

II. Background

A. Limitations of Mutation Analysis

It is infeasible to exhaustively test a test suite, as this would mean running it on all possible programs. Even running it on all mutation operators applied to all statements of a program is non-trivial. This only gets worse as the size and complexity of a code-base grows, leading to a combinatorial explosion [19]. Mutation testing is even costlier for concurrent code because it must be tested against thread schedules and memory reordering.

One way to tackle complexity is reducing the mutants used. Kintis et al. [26] defined the notion of disjoint mutants, i.e., a set of mutants that subsumes all the others. Ammann et al. showed that minimal mutants [4] reduce the count. Kaminski et al. [24, 25] and Just et al. [23] used fault-based predicate testing to reduce redundancy of relational and logical operators. Sampling [40] and searching higher order mutants [17, 18, 27] also helps.

Mutation can leave semantics unchanged. These “equivalent mutants” skew results, and full detection is undecidable [9]. Manual equivalence inspection [46] requires significant effort and provides an identification rate of only 8% [1]. Automated equivalence heuristics are thus attractive, and useful solutions have been proposed [35]. One approach is to use the compiler to detect equivalent mutants [8]. If the original program and a mutant compile to the same object code, no test could reveal a difference. Mothra et al. [35] showed that this identifies about 45% of equivalent mutants. Recently Papadakis et al. proposed Trivial Compiler Equivalence (TCE) which groups mutants with identical object code into equivalence classes [42], addressing the “duplicate mutant” problem. We used TCE to identify both equivalent and duplicate mutants.

B. Read Copy Update (RCU)

The RCU module of the Linux kernel is a synchronization mechanism that allows lightweight readers [30]. RCU read-side critical section entry/exit overhead can be exactly zero [31], excellent for read-mostly workloads [16, 30, 33]. However, RCU updaters cannot exclude readers, and must take care to avoid disrupting readers. Updaters typically maintain versions of the part of the structure being updated, reclaiming old versions only when safe.

RCU use in the Linux kernel has gone from 0 in 2002 to over 6,500 calls in 2013 [32]. RCU pervades the kernel, with one in every 2,000 lines using an RCU primitive [32]. Given the complexity of the code, and its importance, researchers have applied model checking techniques to RCU [2, 15, 29], and the RCU test harness (rcutorture) is very well developed.

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. [14]), 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 [34]. Interest in rcutorture has grown, with the number of contributors growing from 5 to 9 between 2006 and 2014.

Time dependent and stochastic testing systems such as rcutorture are common for critical systems code [38]. The longer you run rcutorture, the higher the chances of finding a bug, if present. This means that non-trivial mutants need to be run for very long periods of time. 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.

III. METHODOLOGY

A. Mutation Generation

We used the tool developed by Andrews et al. [6] to generate mutants. We decided to use this tool as it was evaluated on a set of eight well-known subject programs, part of a Siemens suite [6]. The tool is also simple in design and implementation; a 350 LOC Prolog program and a shell script. This tool generates mutants from a source file, treating each line of code in sequence and applying four classes of “mutation operators”. Every valid application of a mutation operator to a line of code results in a mutant being generated in a separate file. The four classes of mutation operators are given in table I.

TABLE I. MUTATION OPERATORS

Name	Description
rep_const	Replace 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
Negate	Negate the decision in an “if” or “while” statement
del_stmt	Delete a statement

The first three classes are considered "sufficient" mutation operators (i.e., a set S of operators such that test suites that kill mutants in S tends to kill mutants formed by a broader set) [36]. The fourth operator handles pointer-manipulation and field-assignment statements that are not vulnerable to any of the sufficient mutation operators [5]. Table II contains some sample mutants from RCU and table III contains the details of mutants for each mutation operator category.

TABLE II. MUTATION EXAMPLES FROM RCU

Name	Original Version	Mutated Version
rep_const	if (rnp->qsmask ==1)	if(rnp->qsmask !=1)
rep_op	for (i = 0; I >= RCU_NEXT_SIZE; i++)	for (i = 0; i== RCU_NEXT_SIZE; i++)
Negate	if (rcu_batch_empty(b))	if(!(rcu_batch_empty(b)))
del_stmt	struct rcu_head *head;	

After applying the mutation generator to each of RCU’s files (less than 5 minutes for all files), the next step was to compile the 3,169 resulting mutated versions of RCU. For scalability reasons, we did this and all stress testing on virtual machines built on the ESXi 5.5 platform [11].

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 virtual machines. We used 4 virtual machines running in parallel, each of which had 2x 2.7GHz CPUs (x86_64 architecture), with 2 threads per CPU, and 4 GB memory. We integrated RCU with Linux kernel version 3.18.5.

B. Reducing The Test Space

We had to reduce the number of mutants as much as possible and as early as possible. We trivially discarded the 354 (11.1%) which failed to build (mutation tools sometimes produce code which is syntactically nonsensical, e.g. changing parameters to a function call). Next we compared each mutants object code against that of the original code (to identify equivalent mutants) and to that of every other mutant (to identify duplicate mutants).

C. *Running rcutorture On Mutants*

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.

D. *Analysis*

Once the 2 minutes were up we parsed the logs generated by rcutorture for strings like "Assertion failure", "Badness", "WARNING:", "BUG", "!!!," etc. These are coded into the Linux kernel and rcutorture to indicate a failure. We treated mutants triggering such warnings as killed, and mutants that did not generate any warnings as surviving. The only exception was when a mutant caused the kernel to fail to execute.

While we expected to run the surviving mutants with longer and longer test durations, the list of surviving mutants was so small that manual inspection could be performed, which suggests that given a good testing framework like rcutorture, inspecting and checking surviving mutants (and determining true survivors) may be less onerous than expected.

We compiled the list of mutants and sent them to a human "oracle" (a maintainer of RCU and co-author of this paper) for inspection. The oracle examined each surviving mutant to determine if there was a test that would eventually catch the mutant, or whether there was a gap in the test harness. When deficiencies were identified, new tests were built, and the RCU was tested to determine if the gap was masking a bug.

IV. Result

A. *Mutant Attrition*

TABLE III. Mutants in Mutation Operator Category

File	del_stmt	negate	rep_const	rep_op
srcu.c	116	17	72	45
tiny.c	86	12	47	37
update.c	126	25	131	61
tree.c	858	173	732	631
Total	1,186	227	982	774

Figure 1 shows the percentage of mutants that survived the build process by file. We see that invalid mutants were relatively evenly distributed across the 4 files.

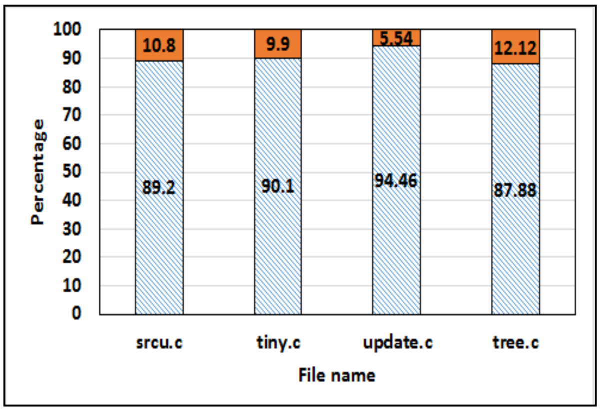

Fig. 1. % of mutants failing/surviving build process (Fail: top of bars)

Applying the TCE, we found that about 70% of buildable mutants were unique (Figure 2). Surprisingly we found a disproportionate number of equivalent in update.c. Of the 2,815 total buildable mutants, 2,150 were unique.

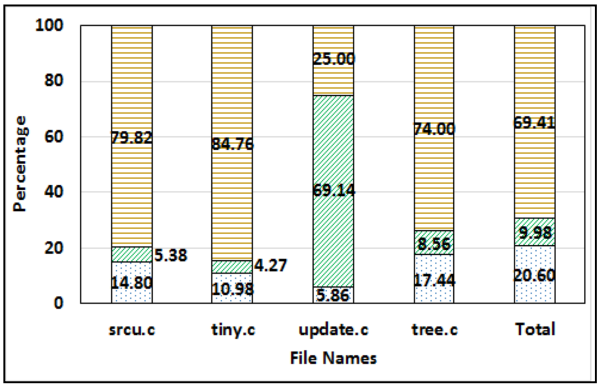

Fig. 2. % of equivalent, duplicate and unique mutants in build surviving mutants (Top: unique, middle: equivalent, bottom: duplicate)

Next we ran our 2-minute test runs of rcutorture on all unique mutants. We found that only 380 mutants (17.7% of unique buildable mutants, 12.0% of generated mutants), survived (not identified as bugs by the test harness). Figure 3 shows the attrition of mutants for each process stage. These were passed on to our human oracle for manual inspection. After manual inspection, our oracle identified 3 weaknesses in rcutorture. Of the 3 failures, 2 were determined to conceal bugs in RCU itself (see Section V).

B. *Time investment*

Generating mutants was trivial, and took on the order of ~150 seconds. It took ~30 minutes to compile each mutated version of the Linux kernel on the machine we used. This process can be parallelized (up to one kernel build per machine/VM). We used the diff command to identify duplicates, which took ~1 second to calculate each diff.

$$N + \sum_{i=1}^{4} n_i^2 \qquad (1)$$

Equation (1) calculates the number of diffs performed, where N is the number of mutants and n_i is the number of mutants in each file (each mutant has to be compared to the gold standard, then to each other mutant).

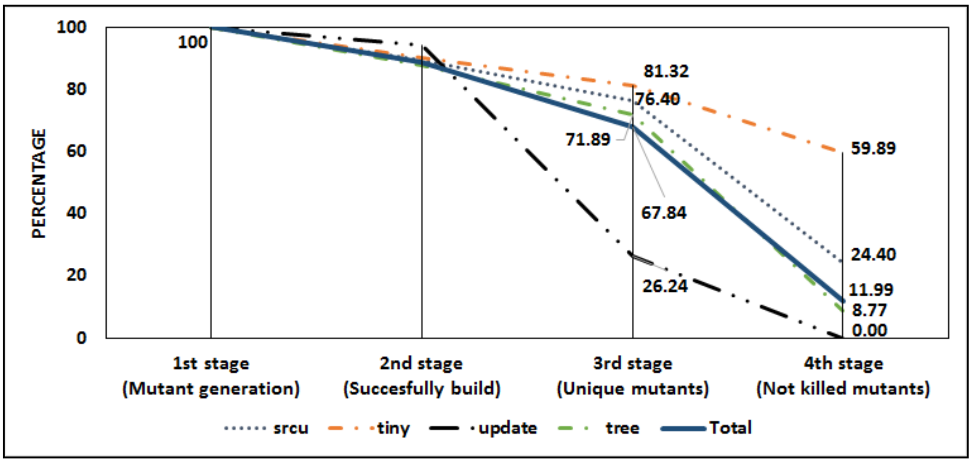

Fig. 3. Percentage of mutant surviving after every stage of processing

Each of the 2-minute rcutorture test runs had to go through a setup phase, generating a set of scripts and building an image of the kernel with a specific configuration to load in Qemu. This one-time setup took ~30 minutes, which preceded each of the 2 minute runs. These images can be reused for longer runs.

Finally, there is the human time investment. Estimating this is harder, since the analysis was performed on a catch-as-catch-can basis as new results arrived. A good approximation is 5 minutes per-mutant, but with very large variance. Some mutants were automatically understood as of no interest, while others required much more effort to analyze (but these also included the most beneficial, the ones resulting in patches). A good estimate for overall human effort is 25 hours.

V. 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. All patches can be accessed using the provided footnotes.

Patch 1: rcutorture: Test SRCU cleanup code path.

Details: 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 structures[1].

Patch 2: rcutorture: 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[2].

Patch 3: rcu: 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[3].

Patch 4: rcu: Don't redundantly disable irqs in rcu_irq {enter,exit}()

This replaces a local_irq_save() and local_irq_restore() pair with a lockdep assertion which removes the corresponding overhead from the interrupt entry/exit fast paths. This change was introduced because mutation testing showed that removing rcu_irq_enter()'s call to local_ird_restore() had no effect, indicating interrupts were always enabled[4].

Patch 5: rcu: Make rcu_gp_init() bool rather than int

Mutation testing showed that the return value from rcu_gp_init() is always used as a boolean, so this commit makes it a Boolean[5].

VI. Discussion

Following the above process, we were able to narrow 3,169 mutants to only 380 potentially interesting mutants with little or no human intervention, using modest compute resources (3,499 hours of runtime on a normal machine, a load which is very parallelizable). While 380 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.

[1] http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=ca1d51ed9809a99d71c23a343b3acd3fd4ad8cbe

[2] http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=f13bad9042dcf9b60b48a0137951b614a2ee24b

[3] http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=6e91f8cb138625be96070b778d9ba71ce520ea7e

[4] http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=7c9906ca5e582a773fff696975e312cef58a7386

[5] http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=45fed3e7cfb4001c80cd4bd25249d194a52bfed3

We found that code that calculates heuristics and error-recovery timeouts can be surprisingly robust to mutations, and adding tests that kill these mutants would lead to more false positives under heavy load or other extreme conditions. Similarly, mutants that cause small degradations in throughput or real-time response may prove difficult to kill. Finally, test suites for algorithms with some degree of redundancy may find it difficult to kill mutants that disable only a subset of the redundant code paths. For example, RCU has a number of quiescent states, including the context switch, the idle loop, usermode execution, and offline CPUs. A mutant that disables detection of any one type of quiescent state will likely survive testing because one of the other types of quiescent states will likely be encountered sooner rather than later.

Bugs found using rcutorture are often non-deterministic. Some may occur only after extremely long runtimes (~1,000 hours). To obtain perfect confidence, rcutorture needs to run for a very long time, which is impractical. Instead, the approach we advocate is to narrow the field of candidates so that either enough machine resources are available, or a human oracle can reasonably inspect and evaluate each case. Our goal is to determine how the set of mutants is further narrowed by longer and longer runtime windows. 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.

Given the complexity of RCU, one could expect to see most mutants fail during compilation. However, only 11% 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'd expect to see an even lower failure rate.

One might hope that a perfect test suite would kill all mutants, but that is unlikely. First is the issue of equivalent mutants. Though we tried to factor most of these out, some cannot be caught by any automated method. For instance, it is common in C to use an integer as a boolean, where 0=false and any non-zero value =true. Mutating one non-zero value to another non-zero value will result in an equivalent program which cannot be detected using diff, depending on how the value is used, and which cannot be killed by a valid test case (due to no semantic difference).

We found that about 10% of our mutants were equivalent, which is close to the findings of Papadakis et al. [42] when they looked at 18 programs. We found that about 20% of the mutants were duplicate mutants, which is also close to their findings. When we look at unique mutants in each file we see that tree.c has the highest percent of unique mutants (74%). This is the biggest file, with 101 functions. tree.c implements a large part of RCU's synchronization.

Any mutant affecting a portion of the program that is conditionally compiled out will "survive," as it is not present in the object code. This usually indicates that the test suite needs to be expanded to include a configuration that compiles and tests the code affected by such a mutant. Similarly, a mutant affecting dead code will survive, but also indicates that the test suite's coverage needs to increase, for example, by including a greater variety of inputs, or, that the code should be removed. In the case where a greater variety of input is required, some sort of randomized testing (e.g., as provided by American Fuzzy Lop (AFL) [3]) can be useful. These last categories of mutants are normally the most productive in terms of improving the test suite. For example, the rcutorture tests for Tiny RCU failed to test callback handling. Fixing rcutorture to cover callback handling by applying patch 2 located a bug in callback handling which was later fixed by applying patch 3.

VII. Threats to validity

We used the tool by Andrews et al. [6] 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.

Other threats are due to the use of potentially faulty software. We used gcc to identify equivalent mutants, but the gcc compiler and diff utility may have defects. However, these systems are heavily tested and deployed, so it is unlikely that they would have such grave defects as to influence our results. We used nested virtualization and that might impact the performance of the guest kernels, but not rcutorture.

VIII. Conclusion and Future work

The main contribution of the paper is an investigation of how to apply mutation analysis on a complex software system, as well as demonstrating the value of doing so, even on well-tested systems. While mutation testing can generate a lot of random results, this randomness can be quickly and efficiently triaged, and a human oracle can concentrate on a small number of interesting cases. We found that mutation analysis can uncover interesting instances of weak testing, even in a robust system like rcutorture. While a fairly large number of mutants were left alive after our initial run, subsequent runs should further reduce the surviving mutants.

References

[1] Acree Jr, & Troy, A. (1980). "On Mutation (No. GIT-ICS-80/12)". In PhD Thesis, Georgia Institute of Technology, Atlanta, Georgia.

[2] Alglave, J., Maranget, L., & Tautschnig, M. (2014). "Herding cats: Modelling, simulation, testing, and data mining for weak memory". In Transactions on Programming Languages and Systems (TOPLAS), Vol. 36, No. 2, (pp. 7).ACM.

[3] American Fuzzy Lop (AFL): http://lcamtuf.coredump.cx/afl/

[4] Ammann, P., Delamaro, M. E., & Offutt, J. (2014, March). "Establishing theoretical minimal sets of mutants". In Seventh International Conference on Software Testing, Verification and Validation, (pp. 21-30). IEEE.

[5] Andrews, J. H., & Zhang, Y. (2003). "General test result checking with log file analysis". In Transactions on Software Engineering, Vol. 29, No.7, (pp. 634-648). IEEE

[6] Andrews, J. H., Briand, L. C., & Labiche, Y. (2005). "Is mutation an appropriate tool for testing experiments?[software testing]". In Proceedings of 27th International Conference on Software Engineering, (pp. 402-411). IEEE.

[7] Baker, R. J., & Habli, I. (2013). "An empirical evaluation of mutation testing for improving the test quality of safety-critical software". In Transactions on Software Engineering, , Vol. 39, No.6, (pp. 787-805). IEEE.

[8] Baldwin, D., & Sayward, F. (1979). "Heuristics for Determining Equivalence of Program Mutations". In PhD Thesis, Georgia Institute of Technology, Atlanta, Georgia..

[9] Budd, T. A., & Angluin, D. (1982). "Two notions of correctness and their relation to testing". In Acta Informatica, Vol. 18, No.1, (pp. 31-45).

[10] Daran, M., & Thévenod-Fosse, P. (1996). "Software error analysis: a real case study involving real faults and mutations". In Software Engineering Notes, Vol. 21, No.3, (pp. 158-171). ACM.

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

[12] Feitelson, D. G. (2012). "Perpetual development: a model of the Linux kernel life cycle". In Journal of Systems and Software, Vol. 85, No.4, (pp. 859-875).

[13] Frankl, P. G., Weiss, S. N., & Hu, C. (1997). "All-uses vs mutation testing: an experimental comparison of effectiveness". In Journal of Systems and Software, Vol. 38, No. 3, (pp. 235-253).

[14] 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.

[15] Groce, A., Ahmed, I., Jensen, C., & McKenney, P. E. (2015)."How Verified is My Code? Falsification-Driven Verification." In Proceedings of the 30th international conference on Automated software engineering.IEEE.

[16] 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).

[17] Harman, M., Jia, Y., Reales Mateo, P., & Polo, M. (2014). "Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation". In Proceedings of the 29th international conference on Automated software engineering, (pp. 397-408). ACM.

[18] Jia, Y., & Harman, M. (2008, September). "Constructing subtle faults using higher order mutation testing". In 8th International Working Conference on Source Code Analysis and Manipulation, (pp. 249-258). IEEE.

[19] Jia, Y., & Harman, M. (2011). "An analysis and survey of the development of mutation testing". In Software Engineering Transactions, Vol. 37, No. 5, (pp. 649-678). IEEE.

[20] Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). "Why don't software developers use static analysis tools to find bugs?". In Proceedings of the International Conference on Software Engineering, (pp. 672-681). IEEE.

[21] Just, R., Ernst, M. D., & Fraser, G. (2014). "Efficient mutation analysis by propagating and partitioning infected execution states". In Proceedings of the International Symposium on Software Testing and Analysis, (pp. 315-326). ACM.

[22] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., & Fraser, G. (2014). "Are mutants a valid substitute for real faults in software testing?". In Proceedings of the 22nd International Symposium on Foundations of Software Engineering, (pp. 654-665). ACM.

[23] Just, R., Kapfhammer, G. M., & Schweiggert, F. (2012). "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis". In Software Reliability Engineering, (pp. 11-20). IEEE.

[24] Kaminski, G., Ammann, P., & Offutt, J. (2011)." Better predicate testing". In Proceedings of the 6th International Workshop on Automation of Software Test, (pp. 57-63). ACM

[25] Kaminski, G., Ammann, P., & Offutt, J. (2013). "Improving logic-based testing". In Journal of Systems and Software, Vol. 86, No. 8, (pp. 2002-2012).

[26] 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.

[27] Langdon, W. B., Harman, M., & Jia, Y. (2010). "Efficient multi-objective higher order mutation testing with genetic programming". In Journal of systems and Software, Vol. 83, No. 12, (pp. 2416-2430).

[28] Liang, L., McKenney, P. E., Kroening, D., & Melham, T. (2016). "Verification of the Tree-Based Hierarchical Read-Copy Update in the Linux Kernel".In arXiv preprint arXiv:1610.03052.

[29] Lissy, A., Laurière, S., & Martineau, P. (2011). "Verifications around the Linux kernel". In Linux Symposium, (p. 37).

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

[31] McKenney, P. E., & Slingwine, J. D. (1998). "Read-copy update: Using execution history to solve concurrency problems". In Parallel and Distributed Computing and Systems, (pp. 509-518).

[32] McKenney, P. E., Boyd-Wickizer, S., & Walpole, J. (2013). "RCU usage in the Linux kernel: one decade later".

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

[34] Nested Virtualization: https://msdn.microsoft.com/en-us/virtualization/hyperv_on_windows/user_guide/nesting

[35] Offutt, A. J., & Craft, W. M. (1994). "Using compiler optimization techniques to detect equivalent mutants". In Software Testing, Verification and Reliability, Vol. 4, No. 3, (pp-131-154).

[36] 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).

[37] Offutt, A. J., Pan, J., Tewary, K., & Zhang, T. (1996). "An experimental evaluation of data flow and mutation testing". In Softw., Pract. Exper, Vol. 26, No. 2, (pp. 165-176).

[38] Pacheco, C., & Ernst, M. D. (2007). "Randoop: feedback-directed random testing for Java". In Companion to the 22nd conference on Object-oriented programming systems and applications, (pp. 815-816). ACM.

[39] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., & Muller, G. (2011). "Faults in linux: ten years later". In Computer Architecture News, Vol. 39, No. 1, (pp. 305-318). ACM.

[40] Papadakis, M., & Malevris, N. (2010). "An empirical evaluation of the first and second order mutation testing strategies". In Third International Conference on Software Testing, Verification, and Validation Workshops, (pp. 90-99). IEEE.

[41] Papadakis, M., & Malevris, N. (2010). "Automatic mutation test case generation via dynamic symbolic execution". In 21st international symposium on Software reliability engineering, (pp. 121-130). IEEE.

[42] Papadakis, M., Jia, Y., Harman, M., & Le Traon, Y. (2015). "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique". In 37th IEEE International Conference of Software Engineering, (pp. 936-946). IEEE.

[43] Read-copy-Update ((RCU): https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt

[44] Smith, B. H., & Williams, L. (2009). "On guiding the augmentation of an automated test suite via mutation analysis". In Empirical Software Engineering, Vol. 14, No. 3, (pp. 341-369).

[45] Wong, W. E., & Mathur, A. P. (1995). "Reducing the cost of mutation testing: An empirical study". In Journal of Systems and Software, Vol. 31, No. 3, (pp. 185-196).

[46] Yao, X., Harman, M., & Jia, Y. (2014). "A study of equivalent and stubborn mutation operators using human analysis of equivalence". In Proceedings of the 36th International Conference on Software Engineering, (pp. 919-930). ACM.

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

[48] Zhang, L., Marinov, D., & Khurshid, S. (2013). "Faster mutation testing inspired by test prioritization and reduction". In Proceedings of the International Symposium on Software Testing and Analysis, (pp. 235-245). ACM