Comparing Mutation Testing at the Levels of Source Code and Compiler Intermediate Representation

Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood, Darko Marinov
Department of Computer Science, University of Illinois at Urbana-Champaign, IL 61801, USA
{hariri2,awshi2,wvf2,msm6,marinov}@illinois.edu

***Abstract*—Mutation testing is widely used in research for evaluating the effectiveness of test suites. There are multiple mutation tools that perform mutation at different levels, including traditional mutation testing at the level of source code (SRC) and more recent mutation testing at the level of compiler intermediate representation (IR).**

This paper presents an extensive comparison of mutation testing at the SRC and IR levels, specifically at the `C` programming language and the LLVM compiler IR levels. We use a mutation testing tool called SRCIROR that implements conceptually the same mutation operators at both levels. We also employ automated techniques to account for equivalent and duplicated mutants, and to determine minimal and surface mutants. We carry out our study on 15 programs from the Coreutils library. Overall, we find mutation testing to be better at the SRC level: the SRC level produces much fewer mutants and is thus less expensive, but the SRC level still generates a similar number of minimal and surface mutants, and the mutation scores at both levels are very closely correlated. We also perform a case study on the *Space* program to evaluate which level's mutation score correlates better with the actual fault-detection capability of test suites sampled from *Space*'s test pool. We find the mutation score at both levels to *not* be very correlated with the actual fault-detection capability of test suites.

I. Introduction

Evaluating the effectiveness of a test suite is a challenging task. A test suite that has a large number of tests, or achieves a high statement or branch coverage, does not necessarily have a high fault-detection capability. A test suite can achieve a high coverage without exercising some particular paths that reveal faults in the code under test, and even if the test suite exercises the faulty paths of interest, it may still not detect the faults if the test oracle does not capture the intended behavior.

Mutation testing is widely used in research for evaluating the quality of test suites. Mutation testing proceeds in two phases. In the first phase, mutation testing generates mutants by applying mutation *operators*, which create simple syntactic changes, to the code under test. Researchers have proposed mutation operators; some of them are general [33], e.g., replacing '+' with '−', and other ones are domain specific, e.g., special mutation operators for web services [41] or for security purposes [39], [40]. We consider only first-order mutation where each mutant has only one change [31]. In the second phase, the test suite is run on each mutant. If at least one of the tests fails on a mutant, the mutant is said to be *killed*. The *mutation score* is the ratio of the *killed* mutants over the total number of mutants run, and a higher mutation score indicates a better test suite. Ideally, the mutants run should not be equivalent to the original code under test (those mutants cannot be killed) or duplicate of one another (those mutants could skew the score).

Multiple mutation tools were developed that perform mutation at different levels, including traditional mutation testing at the level of source code (*SRC*), e.g., for the `C` language [8], [9], [18], [19], [26], [30], [32], and `Java` [34], [43], and mutation testing at the level of compiler intermediate representation (*IR*), e.g., for the LLVM IR [5], [28], [29], [49], [55], [56], [60], and `Java` bytecode/IR [3], [53]. Applying mutation at different levels offers different advantages and disadvantages. First, the mutation testing time is dominated by the time to run the test suite on each mutant, so generating fewer mutants leads to faster mutation testing. Second, the quality of the mutation score is related to the quality of the generated mutants; if a level generates many redundant mutants, the mutation score would be misleadingly high or low. Third, while mutation testing is most commonly used to evaluate test suites through mutation scores, another use case is to reason about individual mutants to improve the test suite to kill said mutants. While comparing the quality of different test suites using mutation scores can be done with either level, reasoning about how to kill individual mutants is easier at the SRC level, due to the changes happening on source code developers wrote themselves [61]. Lastly, mutating at the SRC level requires a separate mutation tool for each programming language. In contrast, a mutation tool at the IR level can be applied to multiple source languages that compile to the same compiler IR. For example, the LLVM IR [46] is a popular IR that supports multiple languages, e.g., `C`, `C++`, `Objective-C`, `Objective-C++`, `OpenMP`, `OpenCL`, and `CUDA`.

We present an extensive comparison of mutation testing at the SRC and IR levels for the `C` language and the LLVM compiler IR. We use our publicly available mutation testing tool SRCIROR [28]. SRCIROR conceptually implements the same mutation operators at both levels: AOR, LCR, ROR, and ICR. A similar set of mutation operators is often used in the existing mutation tools for the `C` language, e.g., by Andrews et al. [8], [9] or Jia and Harman [30], [32]. We perform mutation on all covered code (for all source files) at both levels.

Although we use the same operators at both SRC and IR levels, mutant generation can still lead to different mutants at the two levels. This difference is inherent in the nature of the two levels and in their interplay with compiler optimizations: SRC applies mutations *before* optimizations, and IR applies

mutations *after* optimizations. For example, consider the simple addition statement `z = x + y`. At the SRC level, mutant generation using AOR would replace '+' with other arithmetic operators. In contrast, at the IR level, the compiler may find that `x` and `y` have constant values, and the optimizations (constant propagation and constant folding) would eliminate the add operation, so no AOR mutation would apply. On the other hand, if this statement was in a loop, then loop unrolling could make two copies of this statement, and mutating any one of the two copies at the IR level would not generate the same mutant as at the SRC level (that effectively mutates both copies at once). Our goal is *not* to generate exactly the same set of mutants at both levels; while in principle one could engineer tools to do exactly that, there would be nothing to compare among the resulting set of mutants. Our goal is to compare the two levels with conceptually same mutation operators applied in a natural manner at each level.

We also employ automated techniques to account for equivalent and duplicated mutants, and to determine representative mutants. We statically determine which mutants must be equivalent and duplicated using the trivial compiler equivalence (TCE) technique proposed by Papadakis et al. [51]. We dynamically determine which mutants may be equivalent using a relatively large test pool for each program. We also determine representative mutants using both minimal mutant sets [7] and surface mutant sets [25].

We perform our study on 15 Coreutils programs. Coreutils is a library of command-line Unix utilities written in `C`. The source distribution includes a regression test suite for many of those programs. In total, we carry out our study on 1022 tests. We run all our mutation testing experiments within Docker containers [2], with one Docker container for each program, allowing us to isolate the mutation testing experiments between the different programs and also to ensure that running the different mutants does not have a negative impact on the system running the experiments, e.g., removing files from the file system or changing all the permissions.

We also perform a case study on the widely studied *Space* program. *Space* is developed by the European Space Agency and is publicly available in the Software Infrastructure Repository (*SIR*) [21], along with its large pool of tests and a number of faulty versions. We perform mutation testing on *Space* at both the SRC and IR levels and run its tests on the provided faulty versions (representing real faults) that come with it. We then compare the mutation scores of the two levels computed on test suites constructed from the overall pool of tests to the fault-detection capability of those same test suites.

In our study, we address the following research questions:

- **RQ1.** How does the number of generated mutants differ between the SRC and IR levels?
- **RQ2.** How does the number of equivalent and duplicated mutants differ between the SRC and IR levels?
- **RQ3.** How does the mutation score differ between the SRC and IR levels?
- **RQ4.** How do the mutation score and the number of equivalent and duplicated mutants of different mutation operator classes differ between the SRC and IR levels?
- **RQ5.** Which of the two levels (SRC or IR) generate more minimal and surface mutants?
- **RQ6.** How do mutation scores at the SRC and IR levels compare with the fault-detection capability of test suites?

The summary of our findings is as follows. First, the *total* number of generated mutants and the number of non-equivalent and non-duplicated mutants are *lower* at the SRC level than at the IR level. Second, the mutation scores at the SRC and IR levels are *highly correlated*. Third, the summary conclusions comparing SRC and IR for all operators combined mostly apply to individual mutation operators. Fourth, the SRC and IR levels generate a similar number of minimal and surface mutants. Fifth, based on our case study on *Space*, the mutation scores at both levels are *not* correlated with the actual fault-detection capability of test suites.

Overall we find that mutation testing at the SRC and IR levels are more similar than dissimilar, but with the SRC level offering more advantages than the IR level. As a result, we recommend that researchers apply mutation testing at the SRC level rather than at the IR level, although the SRC level requires implementing a mutation tool for each source programming language.

II. Generating Mutants with SRCIROR

We use SRCIROR [5], our publicly available tool for performing mutation testing for the `C` language at both the source code level and the LLVM intermediate representation (IR) level [5], [28]. SRCIROR provides the same mutation operators at both source code and IR levels, allowing a fairer comparison of mutation testing between the two levels.

A. Mutation Operators

We use four classes of mutation operators in SRCIROR that work on both the SRC and IR levels. These are common mutation operators used in prior work on mutation testing [8], [9], [30], [32]. The four classes of operators are AOR, LCR, ROR, and ICR.

AOR replaces every arithmetic operator from the set {`+`, `-`, `*`, `/`, `%`} with another *different* arithmetic operator from the same set, creating a new mutant with each replacement; at the SRC level, the AOR class also includes replacing the arithmetic assignment operators {`+=`, `-=`, `*=`, `/=`, `%=`}, which does not apply at the IR level where such assignment operators are already translated into simpler instructions.

LCR replaces every logical connector with another logical connector. At the SRC level, it replaces every operator from the set of logical operators {`&&`, `||`}, the set of bitwise operators {`&`, `|`, `^`}, and the set of logical assignment operators {`&=`, `|=`, `^=`} with another *different* operator from the same set, creating a new mutant with each replacement. At the IR level, only bitwise operators are applicable, because the other two sets are translated into different instructions (potentially bitwise operators or conditional branches).

ROR replaces every relational operator with another relational operator. At the SRC level, it replaces every operator from the

set of relational operators {`>`, `>=`, `<`, `<=`, `==`, `!=`} with another *different* operator from the same set, creating a new mutant with each replacement. It also replaces boolean conditions in conditional statements and loops with their negations, specifically it replaces `e` with `!e` for every expression from the set {`if(e)`, `while(e)`, `for(...;e;...)`}. At the IR level, these mutations correspond to replacing every instruction operator from the set {`eq`, `ne`, `ugt`, `uge`, `ult`, `ule`, `sgt`, `sge`, `slt`, `sle`} with a *different* instruction operator from the same set.

ICR replaces every integer constant c with a value from the set {-1, 0, 1, -c, c-1, c+1}\{c}, creating a new mutant with each replacement.

B. Source-level Mutant Generation

SRCIROR generates mutants at the source (SRC) level in the form of a source-to-source transformation tool based on Clang. Furthermore, we configure SRCIROR to collect the coverage of the tests, allowing SRCIROR to only search for candidate mutation locations on the covered lines. SRCIROR applies all mutation operators that are applicable at the covered lines, generating one mutated source file for each mutant. The mutated source file is compiled as usual and linked to produce a final binary executable.

It is important to note that SRCIROR supports mutating multiple files (one per mutant). This is an essential characteristic of a mutation tool, as code is generally organized in multiple files and directories according to its functionality. For example, a significant part of the functionality used by Coreutils tools is defined in files under a utility directory, which get compiled into a shared library `libcoreutils.a`. The shared library is then linked to the executable. As functionality of the code under test is in the shared library, it is important to mutate files from the shared library to ensure the final mutation testing results properly evaluate the quality of the tests.

C. IR-level Mutant Generation

SRCIROR's IR level mutant generation part uses transformation passes in the LLVM compiler infrastructure to generate mutants. SRCIROR generates IR mutants using two LLVM passes. The first pass takes as input a file containing the LLVM IR (also known as *bitcode*) and generates as output the locations that can be mutated and the mutations to apply to each location. The second pass takes as input a file with LLVM IR and the mutation to apply, and then actually applies it. Aside from these passes, SRCIROR includes an LLVM pass that instruments the code and collects coverage at the IR instruction level. Similar to the SRC level mutations, SRCIROR allows the use of the coverage collected in this LLVM pass to restrict the viable locations to mutate.

For our experiments with Coreutils programs, we configure SRCIROR to first instrument the programs and run the tests to collect IR coverage. Then, SRCIROR compiles the source files into LLVM IR files, and takes each IR file to generate a mutated IR file. SRCIROR finally compiles and links each mutated IR file into a final binary executable. Due to having to also compile and link code to create a binary, the cost of generating a single IR mutant is similar to generating a single SRC mutant. This is in contrast to how mutation tools at the IR level for other languages such as `Java` have a much smaller cost for generating mutants, e.g., PIT for `Java` generates mutants dynamically and in-memory through bytecode manipulation [16].

III. Experimental Setup

We describe our experimental setup for comparing mutation testing at the SRC level and at the IR level (answering RQ1 to RQ5). We first describe the programs and their tests we used for the evaluation, along with how we sampled the tests for smaller test suites. We then describe how we determine equivalent and duplicated mutants. We finally describe how we run SRCIROR for both SRC and IR using Docker containers.

A. Evaluation Programs

For our evaluation, we use the programs from Coreutils, a set of programs widely used in empirical evaluations of research in testing [14], [22], [29], [38], [45]. We use the Coreutils version 6.11, because this version is commonly used in previous research and allows for comparison of results across papers. Some of these programs come with manually written tests that invoke the program with different inputs and check expected outputs. Of all the programs from Coreutils, we selected 15 programs for our evaluation. We eliminated the other programs because they had too few tests, because they are not compatible with our Docker running environment (more details later in Section III-D), or because they had flaky tests [42]; a test is flaky if it can pass or fail for the same code under test based on some environment condition that cannot be easily controlled (e.g., the state of the disk, the environment variables, or specific timing/performance issue).

The tests for most programs in Coreutils are manually written scripts that invoke the program multiple times, where each invocation conceptually represents a different test. Such scripts are not ideal for evaluating mutation testing. For example, the program `cut` has a test script file that contains 186 tests. If we were to execute such a test script directly on the original and mutated versions of the program, it would execute *all* 186 tests and report a failure if *any* of the 186 tests fails. Therefore, we would not get the full test-mutant matrix, i.e., we would not know for each test-mutant pair whether that test kills that mutant. If the goal is to evaluate the quality of a test suite, it is enough to know what mutants are killed by any test in the test suite. However, we want to obtain the full test-mutant matrix because it can facilitate a further analysis of mutants, e.g., computation of minimal mutant sets [7]. We use publicly available artifacts from our prior work [28], where we manually analyzed all the test script files for the Coreutils programs and split each long script into several shorter scripts that each runs an individual test.

The number of tests for each program also affects our selection of programs for evaluation. About half the programs have literally no tests or very few tests once skipped tests are

ignored. We ignored those programs because the design of our experiments requires sampling from the entire test pool to form smaller test suites for each program (Section III-B).

B. Sampling Test Suites

When evaluating mutation testing at the SRC level or the IR level, we obtain just one mutation score for the entire test pool of a single program. However, it is difficult to compare the two different levels based solely on just one mutation score. As such, we also sample test suites from the overall test pool for each program, creating a number of smaller test suites for each program. Specifically, from each program's test pool, we sample four different sizes of test suites: 1/2, 1/4, 1/8, and 1/16 of the test pool size. As our smallest size is 1/16 of the test pool size, our criterion then for the number of tests in the test pool for any of the programs we evaluate on is a minimum of 16, to ensure at least one test in a sampled test suite. For each size, we randomly sample the appropriate number of tests from the entire test pool to create a test suite, and we sample ten such test suites per size (with replacement across test suites). We also ensure that no two test suites are equal to each other (although they can have overlaps in tests). We use the same smaller test suites for both SRC level and IR level mutation testing. With the multiple test suites, we can draw correlations between the mutation scores at the SRC and IR levels.

C. Mutant Comparison

We studied mutant generation at the IR level using different optimization levels in our prior work [29]. In this work, we configure both mutant-generation tools to generate mutants using the same optimization level, `-O3`. Furthermore, for both levels of mutant generation, we discard any mutant that fails to compile all the way to binary form.

After generating all the mutants, we determine which mutants are equivalent and duplicated. Following Papadakis et al. [51], equivalent mutants are those that are semantically equivalent to the original binary, while duplicated mutants are those that are semantically equivalent to one another but are not definitely semantically equivalent to the original binary. To automatically determine equivalent and duplicated mutants, we use the *trivial compiler equivalence (TCE)* technique proposed by Papadakis et al. [51] and subsequently used in later work [6], [28], [29], [50]. TCE compares the resulting mutant binaries with the original binary and compares the mutant binaries with one another to determine the equivalent and duplicated mutants. Any mutant binary that is identical to the original binary is an equivalent mutant, and mutant binaries that are identical to one another (but not identical to the original binary) are in the same class of duplicated mutants. To speed up the comparison of these binaries, we compute the checksum of each binary and compare the checksums; specifically, we use `sha1sum` to minimize the chance that different binaries have the same checksum. Note that a given mutant can propagate into multiple executables and/or library files. Similar to our prior work [28], we leverage incremental compilation to only regenerate what is affected by the mutant and checksum only the regenerated files.

It is important to properly handle equivalent and duplicated mutants when computing mutation scores. Equivalent mutants cannot be killed for any test, and duplicated mutants in the same equivalence class should have the same kill or not kill result for any test. As such, keeping equivalent and duplicated mutants can artificially inflate or deflate the overall mutation score. In our analysis of mutation testing, we remove all equivalent mutants and keep only one representative mutant from each equivalence class of duplicated mutants. From our prior work [29], we refer to the resulting mutants as *NEND (not-equivalent, not-duplicated) mutants*.

Once the NEND mutants are determined, one could run the entire mutation analysis only on those mutants *if* all the tests are deterministic. However, in our evaluation, we run all tests on all generated mutants as a means to find *flaky tests*. Flaky test results are unreliable (unless the same test is explored via expensive, multiple runs on the same code [23]), so we cannot easily determine if a mutant is killed or not when the test does not give deterministic results. However, we can determine that we have flaky tests by examining the results of running tests on equivalent and duplicated mutants. If a test kills an equivalent mutant, then the test must be a flaky test. Similarly, if a test kills a mutant from an equivalence class of duplicated mutants but does not kill another mutant from the same equivalence class, then the test is also flaky. By comparing the test results on equivalent and duplicated mutants, we found that tests from some programs (e.g., `join` and `uniq`) have tests that kill equivalent mutants, while tests from some other programs (e.g., `ln`) have tests that do not equally kill all the mutants in the same class of duplicated mutants. As such, we removed several programs from our evaluation.

D. Running Tests on Mutants

The Coreutils programs we use in our evaluation perform functionality that modifies the system underneath. For example, a program from Coreutils that we evaluated on is `chmod`, which changes the permissions of files and directories in the file system. While the manually written tests for `chmod` may be controlled and only change permissions for specific files related to the tests, any mutant created off of `chmod` may not function so cleanly, e.g., modifying the permissions of file to a different set of permissions that are rather hard to recover from. Executing tests on such a mutant can negatively impact the entire system, destroying the environment not just for when doing mutation testing with other programs but also for using the same system again in general. To prevent our mutation testing experiments from affecting each other and the system where we run them, we use Docker [2] to run our mutation testing experiments. We build a basic Docker image that contains Coreutils and SRCIROR along with all the basic dependencies these tools need. We then run mutation testing experiments for each program it its own Docker container built from this basic Docker image, isolating the mutation testing for the mutants of that program from the mutation testing for

TABLE I
NUMBER OF TESTS, FAILED TO COMPILE, SUCCESSFULLY COMPILED, EQUIVALENT, AND DUPLICATED MUTANTS AT BOTH LEVELS FOR EACH PROGRAM

Program	Tests	SRC							IR						
		#F	#M	#E	E%	#D	D%	#NEND	#F	#M	#E	E%	#D	D%	#NEND
chmod	51	269	6546	323	4.9	821	12.5	5402	1627	8288	487	5.9	1199	14.5	6602
cut	186	64	1241	49	3.9	145	11.7	1047	385	3926	165	4.2	524	13.3	3237
dd	16	167	5396	285	5.3	679	12.6	4432	1696	12614	798	6.3	1932	15.3	9884
expr	86	20	535	34	6.4	78	14.6	423	2846	13107	1032	7.9	1741	13.3	10334
factor	31	8	364	12	3.3	37	10.2	315	10	603	66	10.9	62	10.3	475
head	85	47	946	43	4.5	123	13.0	780	385	3642	257	7.1	412	11.3	2973
readlink	159	159	2979	154	5.2	411	13.8	2414	308	3639	288	7.9	542	14.9	2809
seq	37	50	989	40	4.0	123	12.4	826	789	4491	370	8.2	598	13.3	3523
stat	68	64	1815	86	4.7	265	14.6	1464	1102	7162	401	5.6	952	13.3	5809
sum	20	90	2224	121	5.4	300	13.5	1803	39	922	53	5.7	120	13.0	749
tac	52	99	2139	105	4.9	313	14.6	1721	1140	8238	446	5.4	1136	13.8	6656
tail	125	139	3443	193	5.6	447	13.0	2803	1142	7847	410	5.2	1095	14.0	6342
touch	28	129	4763	255	5.4	564	11.8	3944	2076	12662	874	6.9	1744	13.8	10044
unexpand	38	5	343	8	2.3	37	10.8	298	35	1174	52	4.4	212	18.1	910
wc	40	91	2805	140	5.0	382	13.6	2283	2130	11796	603	5.1	1694	14.4	9499
Overall	1022	1401	36528	1848	5.1	4725	12.9	29955	15710	100111	6302	6.3	13963	13.9	79846

the mutants of other programs, as well as isolating the effects of mutation testing runs from the rest of the system.

Given the need for isolation between mutants when running tests on them, it might be ideal to run each mutant and test in their own Docker containers. However, starting up a new Docker container for each test run on a mutant is extremely expensive, especially when computing the full test-mutant matrix as we are doing in this work. As such, for performance reasons, we run all tests for a single project for all its mutants in the same Docker container, but we isolate the mutation testing runs across projects.

However, using a Docker container to run mutation testing for a program is not perfect. Given the nature of the types of tests for Coreutils programs, we find several where the tests actually behave differently in the Docker container and sometimes the tests, even when run on just the unmutated program, cause negative, not recoverable effects in the Docker container. For example, the program `rm` would have tests that create very deeply nested directories that reach the limits of how deep directories can nest. Unfortunately, such a directory structure cannot be deleted in a Docker container (whereas it would be possible in the native system environment), and the program cannot even be rebuilt afterwards due to the creation of such structures. As such, we exclude several Coreutils programs that cannot run properly in a Docker container.

IV. RESULTS AND ANALYSIS

Table I tabulates the programs we use in our evaluation and the number of tests they have.

A. Number of Generated Mutants

Table I also shows the number of mutants that failed to compile (#F), the number of mutants that were compiled successfully (#M). Mutants can fail to compile due to not the mutation tool not having enough context to determine if a mutation is valid or not, e.g., at the SRC level, mutating array initialization such as `int a[10]` with `int a[-1]` would not compile successfully. From the table, we see that there are many more mutants at the IR level that did not compile successfully compared against the SRC level. The percentage of generated mutants that did not compile successfully is 3.7% at the SRC level, versus 13.6% at the IR level.

Table I also shows the number of equivalent (#E) and duplicated (#D) mutants, along with their corresponding percentages computed with respect to the mutants compiled successfully, at each level. We can see that the *total* number of successfully compiled mutants at the IR level is almost three times higher than at the SRC level. The numbers are also higher for the individual programs, as well as for the NEND mutants (removing the equivalent and duplicated mutants).

We next discuss several example mutants from `chmod` to illustrate cases with a one-to-one mapping between SRC and IR mutants, and other cases where one SRC mutant corresponds to multiple IR mutants, and vice versa.

One-to-one mapping between SRC and IR mutants: A mutation applied at the SRC level can in some cases be mapped directly to a mutation at the IR level. For example, in the conditional '`if (optind >= argc)`', ROR at the SRC level replaces the operator '`>=`' with '`>`'. Looking at the IR bitcode, we find the same mutant obtained by replacing the instruction '`icmp sge`' with '`icmp sgt`' also using ROR.

IR presents more mutation opportunities: It is intuitive to expect that IR can present more mutation opportunities than SRC knowing that one SRC statement translates into multiple IR instructions. However, that is not the only reason, and we show here an example not due to the increase in the number of instructions, but due to the more freedom an IR instruction can present in manipulating the functionality. Applying ROR on the line '`if (!change)`' negates the condition of the if statement, namely replaces it with '`if (!(!change))`'. At the IR level, the if statement translates into multiple instructions, including the following integer comparison instruction that checks for equality '`%tobool25 = icmp eq %r* %call24, null`'. The application of ROR replaces '`icmp eq`' with '`icmp neq`' and '`icmp ugt`', generating two mutants at the IR that are similar to the one at

the SRC level. Therefore, the type of the instruction at the IR level enabled more mutants to be generated.

SRC presents more mutation opportunities: In some cases, a SRC statement can present more mutation opportunities than the corresponding IR code. For example, the preprocessed version of function '`mode_changed`' includes this: '`if (new_mode & (04000|02000|01000))`'. Applying ICR at the SRC level generates multiple mutants, e.g., replacing the first occurrence of the constant '04000' with '00'. Note that there are multiple constants that ICR can mutate. Going to the corresponding IR bitcode, the constant folding compiler optimization leads to replacing the entire expression containing multiple constants with just the value '`3584`'. Now there is only one constant that ICR can mutate at the IR level, so there are fewer mutants generated.

Answering **RQ1**, *the number of generated mutants is much higher at the IR level than at the SRC level both before and after taking into consideration equivalent and duplicated mutants; suggesting that mutation testing at the SRC level is much faster to run than mutation testing at the IR level.*

B. Equivalent and Duplicated Mutants

Our results partly agree with those reported in previous work using TCE for equivalent and duplicated mutants [6], [29], [51]. Our experiments show that, on average, 5.1% of the generated mutants at the SRC level are equivalent, while 6.3% of the generated mutants at the IR level are equivalent. For duplicated mutants, the percentage at the SRC level (12.9%) is slightly smaller than at the IR level (13.9%). In general, the percentage of equivalent and duplicated mutants are similar between the two levels. Prior work has reported equivalent mutants ranging between 7% and 7.2%, and duplicated mutants ranging between 13.2% and 21%. Our results show similar ratios. The difference between us and prior work on IR mutation [29] is that first we mutate only covered code, and second we mutate all source files (not just the file containing the main code of the tool).

Answering **RQ2**, *The ratios of both equivalent and duplicated mutants are similar at both the SRC and IR levels. Controlling for equivalent and duplicated mutants is essential to avoid the skewing of mutation scores.*

C. Mutation Score

The mutation score is the main metric in mutation testing, so we compare a variety of mutation scores at the SRC and IR levels. We first compare the mutation scores for only NEND mutants for the entire test pool. We then sample test suites from the test pool, as detailed in Section III-B. For these sampled test suites, we compare the mutation score for both the set of all NEND mutants and the *refined* mutant set [25]. A refined mutant set is a subset of all NEND mutants that are killed by at least one test. Similarly to previous studies [25], [57], [63], we use refined mutant sets to remove the mutants that *may* be equivalent to the original program.

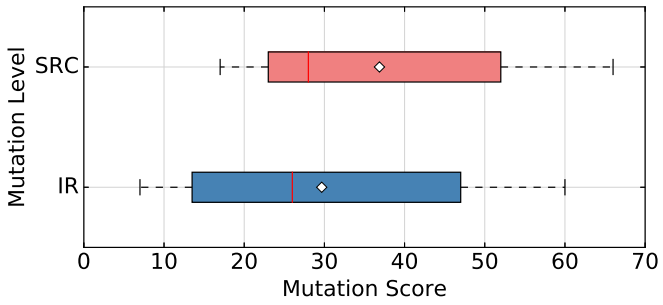

Fig. 1. Distribution of NEND mutation score for the entire test pool at both levels

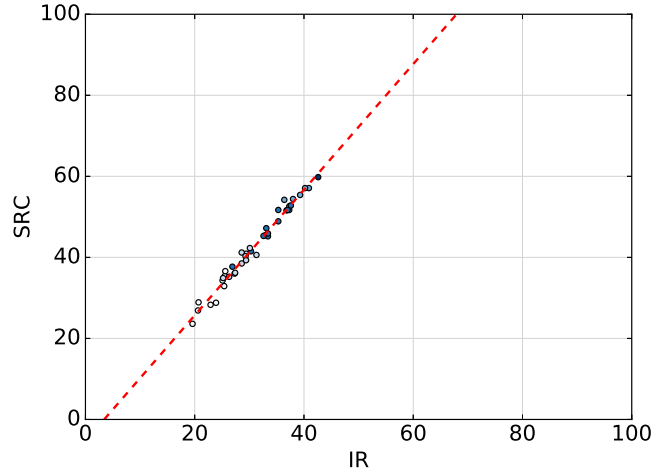

Fig. 2. NEND mutation score for sampled test suites from `seq` at both levels

1) Entire Test Pool: Figure 1 shows in boxplot form the distribution of the NEND mutation scores of all programs for both SRC and IR levels. The median (red line) and (unweighted) mean (white dot) mutation scores are higher for the SRC level compared against the IR level. The weighted mean mutation score (not shown in the figure) at the SRC level is also higher than the one at the IR level, (28.6% vs. 24.1%). Examining individual values and from Figure 1, we can see that the mutation score for both at the SRC level tends to be higher than at the IR level. The Wilcoxon paired rank test for the NEND mutation score shows a p-value of 0.018. Computing the effect size of the difference using Cliff's delta results in a value of 0.28, a small effect size.

2) Sampled Test Suites: We compute the mutation score of sampled test suites of different sizes, considering both all NEND mutants and the refined NEND mutants (when sampling, we only sample from refined NEND mutants w.r.t. the *entire* test suite). Our key goal here is to compare the mutation scores that the sampled test suites obtain at the SRC and IR levels: because most uses of mutation testing in research are to compare test suites, we want to know whether the SRC and IR levels agree on the quality of test suites. If the two levels largely agree, we can use the level that is better by other metrics (e.g., runs faster or is easier to implement).

TABLE II
NUMBER OF GENERATED, EQUIVALENT, AND DUPLICATED MUTANTS ACROSS OPERATOR CLASSES AT THE SRC LEVEL

Program	SRC															
	AOR				LCR				ROR				ICR			
	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND
chmod	811	1.1	6.9	746	608	9.4	0.8	546	3512	5.5	12.8	2871	1615	4.0	1.5	1526
cut	127	0.8	5.5	119	85	7.1	0.0	79	780	4.1	12.1	654	249	4.0	0.4	238
dd	727	2.1	5.4	673	538	8.6	0.7	488	2886	5.8	13.8	2322	1245	4.6	1.5	1169
expr	45	0.0	4.4	43	21	0.0	4.8	20	319	5.3	14.1	257	150	11.3	1.3	131
factor	29	0.0	0.0	29	22	0.0	0.0	22	225	4.0	11.1	191	88	3.4	1.1	84
head	89	1.1	4.5	84	57	14.0	0.0	49	587	5.1	13.6	477	213	1.9	0.5	208
readlink	334	1.8	8.7	299	231	10.8	1.3	203	1765	4.8	13.4	1444	649	6.0	2.2	596
seq	148	1.4	8.8	133	60	0.0	0.0	60	550	6.0	12.0	451	231	2.2	2.2	221
stat	256	1.6	8.6	230	161	9.3	1.2	144	1005	4.9	13.9	816	393	4.6	2.0	367
sum	312	1.3	6.1	289	212	13.7	0.9	181	1225	5.5	14.0	987	475	4.4	1.5	447
tac	299	1.3	7.7	272	174	13.2	1.1	149	1215	4.8	14.1	986	451	4.4	1.1	426
tail	488	3.3	5.3	446	261	11.5	0.8	229	1969	5.2	13.8	1594	725	6.1	1.0	674
touch	606	2.5	6.1	554	443	9.5	0.9	397	2488	5.7	12.7	2031	1226	4.6	1.8	1148
unexpand	20	0.0	5.0	19	14	0.0	0.0	14	226	3.5	11.9	191	83	0.0	0.0	83
wc	359	1.1	7.8	327	245	11.0	0.8	216	1610	5.3	13.5	1308	591	4.1	0.8	562
Overall	4650	1.7	6.6	4263	3132	9.8	0.9	2797	20362	5.3	13.3	16580	8384	4.6	1.4	7880

TABLE III
NUMBER OF GENERATED, EQUIVALENT, AND DUPLICATED MUTANTS ACROSS OPERATOR CLASSES AT THE IR LEVEL

Program	IR															
	AOR				LCR				ROR				ICR			
	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND
chmod	540	0.0	20.6	429	365	2.2	0.3	356	3403	7.9	16.0	2588	3980	5.3	7.4	3476
cut	300	0.0	15.3	254	180	0.0	1.7	177	1822	5.9	14.1	1457	1624	3.5	2.9	1520
dd	995	0.0	18.5	811	500	1.4	1.6	485	4864	7.8	16.7	3675	6255	6.6	6.6	5430
expr	762	0.0	29.3	539	475	3.8	0.4	455	5310	8.5	14.3	4102	6560	8.6	6.7	5556
factor	60	0.0	18.3	49	16	6.2	0.0	15	261	5.7	8.4	224	266	18.8	2.3	210
head	346	0.0	10.7	309	114	0.9	0.9	112	1453	5.2	14.0	1175	1729	10.5	2.8	1499
readlink	184	0.0	25.0	138	146	2.1	0.7	142	1764	8.8	17.6	1298	1545	8.3	5.4	1332
seq	420	0.2	21.2	330	92	0.0	0.0	92	1654	7.0	12.5	1331	2325	10.9	6.1	1931
stat	498	0.0	18.1	408	246	1.6	0.0	242	2700	7.5	15.4	2083	3718	5.2	4.9	3342
sum	132	0.0	7.6	122	32	6.2	3.1	29	342	7.6	14.3	267	416	6.0	4.3	373
tac	654	0.0	15.3	554	282	1.8	0.4	276	3267	7.0	16.0	2513	4035	5.2	4.8	3631
tail	593	0.0	18.7	482	272	1.5	0.4	267	3050	7.1	16.0	2346	3932	4.8	4.8	3553
touch	848	0.0	19.6	682	406	1.0	0.5	400	4825	7.4	15.3	3732	6583	7.8	5.8	5685
unexpand	120	0.0	20.0	96	46	0.0	4.3	44	603	8.1	16.3	456	405	0.7	4.0	386
wc	996	0.0	19.8	799	420	1.2	0.5	413	4378	6.8	15.4	3403	6002	5.0	5.9	5351
Overall	7448	0.0	19.4	6002	3592	1.7	0.7	3505	39696	7.4	15.4	30650	49375	6.7	5.7	43275

We measure correlation using Pearson's R^2, which checks for a linear relationship: a high value would show that the mutation scores at the SRC and IR levels change by a linearly proportional amount (irrespective of the direction).

To visualize the correlation we want to compute between SRC and IR level mutation scores, we show Figure 2, a scatter plot of mutation scores computed on all NEND mutants for one sample program, `seq`. We show this scatter plot for only `seq` as most of the other programs demonstrate a similar trend. Each point in the scatter plot corresponds to a sampled test suite from `seq`. The darker the point color, the bigger the test suite is (the darkest corresponding to the entire test pool and the lightest corresponding to the size 1/16th of the test pool). The x-coordinate is the IR level mutation score, and the y-coordinate is the SRC level mutation score. The strong linear correlation between SRC and IR level mutation scores for `seq` can be seen from the figure, with a high R^2 value (1.0).

We compute theR^2 values comparing mutation scores across levels using all sampled test suites for each program for the set of NEND mutants and the set of refined mutants. The average values of R^2 across all programs is 0.8 for NEND mutants, and 0.8 for refined mutants. We also inspected the individual values per tool and noticed that there is a strong positive correlation between the mutation scores at the SRC and IR levels for most programs. Five programs have R^2 values less than the average value of 0.8 (four programs when considering refined mutants), though all programs have R^2 values of at least 0.6. All R^2 values are statistically significant with $p < 0.001$.

Answering **RQ3**, *mutation scores at the SRC and IR levels are correlated and can be used as good proxies of each other.*

D. Analysis Across Mutation Operators

So far we have analyzed mutation testing by examining the number of generated mutants, the number of NEND mutants, and the mutation score for all four classes of the mutation operators combined. To better understand how mutation testing at the SRC and IR levels compare to each other, we perform

our analysis for each operator class separately, which is a finer granularity that sheds light on whether the results are generalizable or due to a specific mutation operator.

1) Number of Mutants: Tables II and III show the number of mutants generated across different operator classes at the SRC and IR level, respectively. The total number of mutants at the SRC level is lower than at the IR level for every single operator class . Similarly, the number of NEND mutants is lower at the SRC level than at the IR level. For ROR, the SRC level would generate five (likely NEND) mutants replacing each relational operator by a different one from the set {>, >=, <, <=, ==, !=}, whereas IR would generate nine (likely NEND) mutants replacing each operator by a different one from the set {`eq`,`ne`,`ugt`,`uge`,`ult`,`ule`,`sgt`,`sge`,`slt`,`sle`}. The difference of five vs. nine leads to having about twice as many ROR mutants at the IR level than the SRC level. Inspecting the ICR mutants, IR has almost seven times more mutants due to large number of *getelementptr* instruction (which contributes the majority of the ICR mutants) in LLVM used to compute offsets. Another reason that the IR bitcode presents more opportunities is that the compiler introduces more instructions that require address computations, such as replacing array indexing `a[i]` with `a+4*i`.

2) Equivalent and Duplicated Mutants: We look at the percentage of equivalent and duplicated mutants broken down across each mutation operator. We use both the Wilcoxon paired rank test and compute Cliff's delta comparing the percentages of equivalent and duplicated mutants per each operator across all programs between SRC and IR. For equivalent mutants, we observe statistically significant differences ($p < 0.001$ for ROR, with a large Cliff's delta effect size, -0.90. For duplicated mutants, we observe statistically significant differences ($p < 0.001$) for AOR and ICR, with large Cliff's delta effect size values of -0.96 and -1.00, respectively.

3) Mutation Score: Table IV shows the mutation scores of the NEND mutants across mutation operators. The overall mutation scores for the same mutation operators are rather similar between the SRC and IR levels. A Wilcoxon paired rank test comparing mutation scores for each operator between the two levels results in no p-value less than 0.01.

Table V shows p-values for using the Wilcoxon paired rank test comparing number of NEND mutants, percentage of equivalent/duplicated mutants, and mutation scores across operators between SRC and IR levels. The p-values for the tests vary across the different operators.

Answering **RQ4**, *the summary conclusions comparing SRC and IR for all operators combined mostly apply to individual mutation operator and therefore are generalizable*.

E. Minimal and Surface Mutants

So far the SRC level mutation looks better than the IR level mutation, because SRC level generates fewer NEND mutants, and though they result in somewhat higher mutation scores, the mutation scores between the two levels are very correlated. However, these scores could be affected by redundant mutants [7] that do not add any value to the evaluation of test suites. These mutants can misleadingly inflate the mutation score, e.g., some mutants may be easy to kill by any test. We want to find the mutants that are representative of all the other mutants as well as being harder to kill. Following Gopinath et al. [25], [26], we compute two sets of mutants: the *minimal set of mutants* and the *surface set of mutants*[1]. Both sets are defined using dynamically subsuming mutants: a mutant m *subsumes* another mutant m' for a test pool T if every test from T that kills m also kills m'; the subsuming mutant m is a higher quality mutant that is harder to kill. The subsuming mutants are killed by a more diverse set of tests, and the idea is that they are representative of all mutants [49]. The subsuming mutants are considered harder-to-kill than the subsumed mutants [49]. Given a set of mutants, a surface mutant set is a maximal subset that has no subsumed mutants, with subsumption computed over the entire test pool. A minimal mutant set is computed the same, except subsumption is computed over a minimal test suite (i.e., a test suite that is a subset of the test pool, has the same mutation score as the test pool, and if one test is removed, the mutation score drops).

Table VI shows the number of minimal and surface mutants computed for each program. We see that the numbers of minimal and surface mutants are similar at both levels. The main exceptions are `expr` and `tail`, where there are many more minimal and surface mutants at the IR level than at the SRC level for `expr`, while the opposite occurs for `tail`.

TABLE IV
NEND MUTATION SCORE ACROSS MUTATION OPERATOR CLASSES

Program	SRC				IR			
	AOR	LCR	ROR	ICR	AOR	LCR	ROR	ICR
chmod	28.8	35.5	29.3	28.8	31.7	37.4	29.4	32.0
cut	35.3	49.4	53.4	50.0	37.8	40.1	39.1	35.5
dd	16.8	19.3	19.3	14.2	21.0	15.7	17.1	11.2
expr	86.0	50.0	63.4	60.3	76.4	51.6	45.8	54.6
factor	62.1	59.1	49.7	61.9	91.8	86.7	55.4	57.1
head	50.0	24.5	45.9	53.4	35.0	15.2	29.7	23.7
readlink	28.8	12.3	22.8	23.2	18.8	6.3	13.0	13.7
seq	69.2	45.0	61.2	55.2	73.6	30.4	43.6	37.6
stat	32.2	7.6	14.1	23.2	13.0	7.4	9.7	7.4
sum	38.8	17.7	16.4	27.1	82.0	75.9	43.1	58.7
tac	35.7	9.4	21.2	24.9	23.5	6.2	13.5	8.1
tail	34.5	17.5	25.5	30.4	14.7	4.9	9.1	5.7
touch	33.9	16.6	27.6	26.3	27.0	16.0	21.1	17.2
unexpand	73.7	71.4	62.8	68.7	69.8	84.1	48.5	53.1
wc	38.2	25.0	26.0	30.4	31.5	15.5	18.9	15.5
Overall	33.1	22.9	27.9	28.8	34.9	23.3	24.7	22.3

TABLE V
p-VALUES FOR WILCOXON PAIRED RANK TEST COMPARING DIFFERENT VALUES AT SRC AND IR LEVEL

Comparison point	Operators			
	AOR	LCR	ROR	ICR
Number of NEND mutants	0.073	0.191	0.004	< 0.001
% equivalent mutants	0.002	0.003	< 0.001	0.020
% duplicated mutants	< 0.001	0.889	0.006	< 0.001
Mutation score	0.303	0.530	0.025	0.012

[1]There is some terminology mismatch: what Gopinath et al. [25], [26] define as "surface" mutants is what Ammann et al. define as "minimal" mutants [7] and is what Kintis et al. define as "disjoint" mutants [36]. What Gopinath et al. [26] call "minimal" could have been called differently, e.g., "doubly minimal". We follow the terminology from Gopinath et al. [25], [26] as they are the most recent of these papers.

TABLE VI
NUMBER OF MINIMAL MUTANTS AND SURFACE MUTANTS

Program	#Minimal		#Surface	
	SRC	IR	SRC	IR
chmod	17	14	23	16
cut	17	19	20	24
dd	8	10	9	11
expr	19	46	19	47
factor	3	5	4	6
head	15	17	21	22
readlink	10	6	25	10
seq	16	18	17	19
stat	11	9	19	15
sum	7	7	10	9
tac	6	6	8	9
tail	28	9	36	9
touch	15	16	15	16
unexpand	6	7	13	11
wc	12	15	14	17
Overall	190	204	253	241

Performing a Wilcoxon paired rank test between the number of minimal mutants for each program at SRC and IR shows a p-value of 0.6207; the Cliff's delta is 0.04, a neglible effect size. The same tests for surface mutants shows a p-value of 0.9545; the Cliff's delta is 0.14, again, a neglible effect size.

Answering **RQ5**, *the SRC and IR levels generate a similar number of minimal and surface mutants.*

V. CASE STUDY WITH REAL FAULTS

We have found that SRC and IR are good proxies for each other, with the mutation score of SRC being higher than that of IR on average. The next question that arises is: which of the two levels has a mutation score that correlates better to the actual fault-detection capability of the test suite?

To answer this question, we conduct a case study on *Space*, an interpreter of an array definition language (ADL) developed by the European Space Agency [1]. We use the version `2.0`, most recent available version in SIR [4] at the time. It comes with a test pool of 13496 test cases and with 35 documented real faults, with a mapping from tests to faults detected. (The total number of versions documenting real faults that come with *Space* is 38, but only 35 of them are not equivalent to the original program for the given test suite, i.e., the other three are not detected by the given test suite.)

A. Setup

We generate SRC and IR mutants for *Space*, and we run the entire test suite for the NEND mutants. We compare the output (standard output and error) of each test to that we obtain when executing the test on the original program (we compute the checksum of the outputs to compare them). If the output is not the same, we consider the test to have killed the mutant.

Following Andrews et al. [8], we generate 5000 test suites by randomly selecting test suites of size 100 each from the full test pool. We then calculate for each test suite S the mutation detection ratio $Am(S)$, defined as the number of mutants killed by S divided by the total number of mutants, and $Af(S)$ as the number of faults detected by S divided by the total number of faults (35). For each test suite S, we compare $Am(S)$ of SRC and $Am(S)$ of IR to $Af(S)$.

B. Results

We generate a total of 8647 SRC NEND mutants and 22187 IR NEND mutants. We again use Pearson's R^2 to see if there is any linear correlation between SRC and IR $Am(S)$ of the different test suites against each test suite's $Af(S)$. For SRC, the R^2 value is 0.24, $p < 0.001$; for IR, the R^2 value is 0.30, $p < 0.001$. From this value, IR is actually slightly more linearly correlated with a test suite's fault-detection capability, but both levels are not very correlated. To see if there is any correlation, not just linear, we use Kendall's τ_b. Kendall's τ_b reports a τ_b value, which ranges from 0 to 1, with 0 meaning no correlation and 1 meaning a perfect correlation. For SRC, the τ_b value is 0.15, $p < 0.001$; for IR, the τ_b value is 0.19, $p < 0.001$. Once again, IR has a slightly better correlation, but both values are still very low. These statistics suggest that there is very little correlation between mutation score at either level with a test suite's fault-detection capability. The low correlation agrees with recent findings from Papadakis et al. [52], where they find the correlation between mutants killed and fault-detection to drop as test-suite size is controlled.

Answering **RQ6**, *Both levels show very little correlation with the fault detection capability of test suites with IR having a slightly higher correlation than SRC.*

VI. THREATS TO VALIDITY

Our results may not generalize to all software because the programs we chose for our evaluation may not be representative of all software. To address this threat, we used Coreutils, which is commonly used in previous research. We chose a total of 15 programs from Coreutils, i.e., all programs that had a non-trivial number of tests and had tests that were not flaky. For each program, we used all its tests.

In our evaluation, we rely on Clang and LLVM, along with their built-in optimizations, to perform mutation testing at both the SRC and IR levels. Our results may not generalize when compiling with other compilers, such as gcc, which may use different intermediate representations. The evaluation is conducted over four classes of mutation operators at each level. The general results obtained could be specific to those operators and may not generalize to other classes of operators. In fact, repeating our analysis by each class of the mutation operators already shows that the general conclusions can be influenced by some class and need not be similar for each and every class. To determine equivalent and duplicated mutants, we use trivial compiler equivalence (TCE) [51] and we also compare using refined mutants. While TCE finds mutants that are definitely equivalent and duplicated, it gives a lower bound on the actual number of equivalent mutants; in contrast, refined mutants give an upper bound on the actual number of equivalent mutants. The true number of equivalent mutants is between these bounds and could affect our findings of the mutation score. However, detecting all equivalent mutants is an undecidable problem [12], so we use both NEND and refined mutants to compare SRC and IR level mutation.

VII. Related Work

Mutation testing has been widely studied since its introduction [20], [64]. A large number of mutation tools have been introduced for multiple programming languages including `C` [19], [32], `C++` [37], `Java` [43], [44], and many others [13], [15]. Jia and Harman [33] provide a survey of mutation testing.

One area of research on mutation testing has focused on the comparison of mutation testing tools for a single programming language. Multiple studies [17], [25], [58] compared different mutation testing tools for `Java` programs. The work most similar to ours is the work by Gopinath et al. [25]. In their work, they compared three mutation testing tools that work on `Java` applications. Two of the tools used in the study generated mutants on the `Java` byte-code (effectively the IR for `Java`) while the remaining tool generated mutants on the `Java` source code. As such, the authors were able to evaluate the effects of generating mutants at different levels, SRC vs. IR, and they found that the level at which mutants are generated does not significantly affect the mutation score. However, they used the tools out-of-the-box and did not control for mutation operators. In our study, we use mutant generation tools with the same operators at both levels, allowing for a fairer comparison of the effects of mutants generated at different levels. Furthermore, we evaluate with `C` applications. Our findings are similar to theirs in terms of mutation score, but we also study the number of equivalent and duplicated mutants, the breakdown per operator, and multiple types mutation scores at more depth.

In our prior work [29], we studied the effects of compiler optimizations on mutation testing. We mutated IR that was compiled at different optimization levels (`-O0` vs. `-O3`). We later released our tool SRCIROR for performing mutation testing at both the SRC and IR levels [5]. Expanding on our prior work [28], we compare SRC and IR on more Coreutils programs, and we sample the test pools to create smaller test suites to measure the correlation of mutation scores between the two levels. We futher investigated minimal and surface mutants and comparing against faults in *Space*.

Another line of research involves identifying equivalent mutants. The problem is known to be undecidable in general [12]. One common methodology that has been used to detect equivalent mutants is the development of heuristics that can point to likely equivalent mutants [10], [27], [54]. For example, Offutt and Pan formulated the problem of identifying equivalent mutants as a constraint optimization problem [48]. Their approach analyzes the original code and a mutant's path conditions, and it uses those as constraints to determine if the mutant is equivalent to the original code. Another example is the work of Papadakis et al. [51] that evaluated trivial compiler equivalence (TCE) to identify definitely equivalent mutants. They show that a significant portion of equivalent mutants can be detected by comparing the mutated binaries with the original binary. We also use TCE to identify equivalent mutants and compare the number of equivalent mutants at each level. We find that TCE is crucial when a tool mutates all code and not only selected functions [51].

Another similar problem is detecting redundant mutants. Redundant mutants are mutants that map to similar potential faults in the program, such that test suites that kill any of these mutants only can detect the same kind of faults [35]. Having fewer redundant mutants can increase the efficiency of mutation testing, because not all redundant mutants should be run during mutation testing. Gopinath et al. [26] defined both minimal and surface mutant sets, used to identify and avoid redundant mutants. We compare both minimal and surface mutant sets at the two levels, following the terminology from Gopinath et al.. The surface mutants from Gopinath et al. are also defined by Ammann et al. [7] as minimal mutants (different than what Gopinath et al. defined as minimal mutants) and by Kintis et al. [36] as disjoint mutants. Papadakis et al. [49] recently studied surface mutants as part of their work on comparing indicators of mutant quality, classifying surface mutants as mutants that are diversely killed by different tests, being representative of all the other mutants.

Selective mutation aims to select a subset of mutants to be used for evaluation. Most selection methods work by comparing the mutants generated by different mutation operators to identify a subset of the operators that produce high quality mutants [11], [24], [47], [59]. Zhang et al. [63] compared selective mutation based on operators with random mutant selection and concluded that random mutant selection can also give similar results. A more recent study [62] combined operator based mutant selection with random selection by sampling mutants from selected operators and reported that the combination is also effective. We compare mutation testing at the two different levels while breaking down the results based on mutation operator to see if any specific operator has a bigger effect on one level or the other. We find that for the comparison of SRC vs IR, the results for different operators are similar for mutation scores.

VIII. Conclusions

We present the first extensive study that compares mutation testing at the SRC and IR levels. We perform our study on 15 applications from Coreutils programs with 1022 tests. To ensure a fair comparison between the two levels, we use the same mutation operators for SRC and IR.

Our results show that mutation testing at the SRC level is more economical to run than at the IR level. We also find that the NEND mutants at both levels have similar quality and lead to mutation scores that are highly correlated. These results generalize across operators, and are not specific to only some operators. Furthermore, we show the effect of mutating test code on the mutation score. For researchers using mutation testing to compare test suites or testing techniques, we recommend performing mutation testing at the SRC level.

Acknowledgements

We thank Sarfraz Khurshid, Sasa Misailovic, and Tao Xie for their discussions about this project. This research was partially supported by NSF Grant Nos. CCF-1421503, CCF-1629431, CNS-1646305, and CNS-1740916.

REFERENCES

[1] C object biographies. http://sir.unl.edu/portal/bios/space.php.
[2] Docker. https://www.docker.com/.
[3] Real world mutation testing. http://pitest.org/.
[4] Software-artifact infrastructure repository. http://sir.unl.edu/portal/index.php.
[5] SRCIROR. https://github.com/TestingResearchIllinois/srciror.
[6] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce. Evaluating non-adequate test-case reduction. In *ASE*, 2016.
[7] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *ICST*, 2014.
[8] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, 2005.
[9] J. H. Andrews and A. Alipour. MutGen tool. https://github.com/alipourm/cmutate.
[10] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Technical report, DTIC Document, 1979.
[11] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *STVR*, 11(2), 2001.
[12] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Inform.*, 18(1), 1982.
[13] T. A. Budd, R. J. Lipton, R. DeMillo, and F. Sayward. The design of a prototype mutation system for program testing. In *AFIPS*, 1978.
[14] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
[15] W. Chan, S. C. Cheung, and T. Tse. Fault-based testing of database application programs with conceptual data model. In *QSIC*, 2005.
[16] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: A practical mutation testing tool for Java (demo). In *ISSTA*, 2016.
[17] M. Delahaye and L. Du Bousquet. A comparison of mutation analysis tools for Java. In *QSIC*, 2013.
[18] M. E. Delamaro and J. C. Maldonado. Proteum tool for mutation testing of C programs. https://github.com/magsilva/proteum.
[19] M. E. Delamaro and J. C. Maldonado. Proteum—A tool for the assessment of test adequacy for C programs. In *PCS*, 1996.
[20] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4), 1978.
[21] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, 10(4), 2005.
[22] S. Dong, O. Olivo, L. Zhang, and S. Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *ISSRE*, 2015.
[23] M. Gligoric, V. Jagannath, and D. Marinov. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In *ICST*, 2010.
[24] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *ISSTA*, 2013.
[25] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce. Does choice of mutation tool matter? *SQJ*, 25(3), 2016.
[26] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. Measuring effectiveness of mutant sets. In *ICSTW*, 2016.
[27] B. J. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *ICSTW*, 2009.
[28] F. Hariri and A. Shi. SRCIROR: A toolset for mutation testing of c source code and llvm intermediate representation. In *ASE Demo*, 2018.
[29] F. Hariri, A. Shi, H. Converse, S. Khurshid, and D. Marinov. Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level. In *ISSRE*, 2016.
[30] Y. Jia. Milu: A higher order mutation testing tool. https://github.com/yuejia/Milu.
[31] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *SCAM*, 2008.
[32] Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *TAIC PART*, 2008.
[33] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5), 2011.
[34] R. Just. The major mutation framework: Efficient and scalable mutation analysis for Java. In *ISSTA*, 2014.
[35] R. Just, G. M. Kapfhammer, and F. Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *ICST*, 2012.
[36] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *APSEC*, 2010.
[37] M. Kusano and C. Wang. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *ASE*, 2013.
[38] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, 2012.
[39] Y. Le Traon, T. Mouelhi, and B. Baudry. Testing security policies: going beyond functional testing. In *ISSRE*, 2007.
[40] S. Lee, X. Bai, and Y. Chen. Automatic mutation testing and simulation on OWL-S specified web services. In *ANSS*, 2008.
[41] S. C. Lee and J. Offutt. Generating test cases for XML-based web component interactions using mutation analysis. In *ISSRE*, 2001.
[42] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *FSE*, 2014.
[43] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava: a mutation system for Java. In *ICSE*, 2006.
[44] L. Madeyski and N. Radyk. Judy-A mutation testing tool for Java. *IET software*, 4(1), 2010.
[45] P. D. Marinescu and C. Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE*, 2012.
[46] U. of Illinois. The LLVM compilation infrastructure. http://llvm.org/.
[47] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *TOSEM*, 5(2), 1996.
[48] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *COMPASS*, 1996.
[49] M. Papadakis, T. T. Chekam, and Y. Le Traon. Mutant quality indicators. In *ICSTW*, 2018.
[50] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon. Threats to the validity of mutation-based test assessment. In *ISSTA*, 2016.
[51] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE*, 2015.
[52] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae. Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults. In *ICSE*, 2018.
[53] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for Java. In *ESEC/FSE*, 2009.
[54] D. Schuler and A. Zeller. (Un-) Covering equivalent mutants. In *ICST*, 2010.
[55] E. Schulte. llvm-mutate. http://eschulte.github.io/llvm-mutate/.
[56] E. Schulte. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution*. PhD thesis, University of New Mexico, 2014.
[57] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ICSE*, 2008.
[58] P. K. Singh, O. P. Sangwan, and A. Sharma. A study and review on the development of mutation testing tools for Java and Aspect-J programs. *IJMECS*, 6(11), 2014.
[59] B. H. Smith and L. Williams. An empirical evaluation of the MuJava mutation operators. In *TAICPART-MUTATION*, 2007.
[60] M. Sousa and A. Sen. Generation of TLM testbenches using mutation testing. In *CODES+ISSS*, 2012.
[61] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *ICSE*, 2014.
[62] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *ASE*, 2013.
[63] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *ICSE*, 2010.
[64] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4), 1997.