Evaluating the Effects of Compiler Optimizations on Mutation Testing at the Compiler IR Level

Farah Hariri[1], August Shi[1], Hayes Converse[2], Sarfraz Khurshid[2], Darko Marinov[1]
[1] Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801, USA
{hariri2,awshi2,marinov}@illinois.edu
[2] Department of Electrical and Computer Engineering
The University of Texas at Austin, TX 78712, USA
{hayesconverse,khurshid}@utexas.edu

Abstract—**Software testing is one of the most widely used approaches for improving software reliability. The effectiveness of testing depends to a large extent on the quality of test suites. Researchers have developed various techniques to evaluate the quality of test suites. Of these techniques, mutation testing is generally considered to be the most advanced but also expensive. A key result of applying mutation testing to a given test suite is the *mutation score* representing the percentage of mutants killed by the test suite. Ideally the mutation score is computed ignoring the mutants that are semantically equivalent to the original code under test or to one another.**

In this paper, we investigate a new perspective on mutation testing: evaluating how standard compiler optimizations affect the cost and results of mutation testing performed at the compiler intermediate representation. Our study targets LLVM, a popular compiler infrastructure that supports multiple source and target languages. Our evaluation on 18 *Coreutils* programs discovers several interesting relations between the numbers of mutants (including the numbers on equivalent and duplicated mutants) and mutation scores on unoptimized and optimized programs.

Keywords-**mutation testing; compiler optimizations; LLVM**

I. Introduction

Software testing is an important activity for increasing software reliability. Testing is conceptually simple: the code under test is run against a test suite that consists of multiple tests; each test provides some input and checks the actual output against the expected output. However, testing has several challenges in practice. One key challenge is to evaluate the quality of test suites.

Mutation testing is widely used in research to evaluate the quality of test suites, and it is often considered the most powerful approach for this purpose [2], [17]. Mutation testing proceeds in two phases. First, a number of mutants are generated by applying *mutation operators*, which are program transformations that introduce small syntactic changes, to the original code under test. Second, the test suite is run against the mutants to determine which are *killed*, i.e., which mutants give a different output from the original code under test. (More precisely, we are interested in *strong mutant killing*, based on the observable output, rather than weak mutant killing, based on the intermediate state of a test's execution [2].) Finally, the *mutation score* is computed as the ratio of the number of killed mutants to the number of all generated mutants.

While the number of killed mutants depends on the test suite, the number of generated mutants depends on the mutation operators and the level at which the operators are applied. Mutation operators have been proposed for many programming languages, including Ada, C, Cobol, C#, Fortran, Java, and SQL [2]. Mutation testing was also recently applied at the level of compiler intermediate representation (IR) [33]–[35] using LLVM [19], [23]. One IR usually supports multiple source and target languages. For example, LLVM is a widely used compiler infrastructure that supports multiple source languages (including C/C++ via the *Clang* front-end [22]) and multiple target languages (X86-32, X86-64, and ARM). When mutation testing is implemented once at the IR level, it enables mutation testing effectively "for free" for all source languages supported by the IR, without having to implement a special tool for every one of them. However, applying mutations at the IR level means that mutation testing becomes more susceptible to the effects of compiler optimizations.

Compiler optimizations are automatic program transformations applied with the goal of improving some measure of program performance. Compiler optimizations have to preserve the behavior of the code and produce semantically equivalent programs; in contrast, mutation operators aim to produce semantically non-equivalent programs. Some mutants cannot be killed by any test; these mutants are semantically *equivalent* to (albeit being syntactically different from) the original code under test. Determining which mutants are equivalent is undecidable in general [2], [5]. Researchers have proposed several heuristics [1], [4], [11], [13], [15], [25], [27], [28], [31], [32] that help in determining which mutants are more *likely* equivalent or non-equivalent to the original code. Most recently, Papadakis et al. [29] proposed a technique for finding mutants that are *definitely* equivalent to the original code by comparing the compiled versions of the original code and its mutants; their experiments applied the mutation operators on the source code, specifically in the C programming language. They also use the same technique to determine what they call *duplicated mutants* that are definitely equivalent to one another but not definitely equivalent to the original code.

In this paper, we present an empirical study of the effects of compiler optimizations on mutation testing at the compiler IR level. Our study aims to investigate whether mutation testing should be applied with or without compiler optimizations, providing the user with insights about the interplay between mutations and compiler optimizations. To that end, we ask the following research questions:

- **RQ1.** How do compiler optimizations affect the number of generated mutants?
- **RQ2.** How do compiler optimizations affect the number of equivalent and duplicated mutants?
- **RQ3.** How do compiler optimizations affect the mutation score?
- **RQ4.** How do these effects vary with the class of mutation operators applied?

To address these questions, we evaluate several traditional classes of mutation operators (proposed by Offutt et al. [26] for selective mutation). Specifically, we implemented for LLVM the following four classes of mutation operators:

- AOR replaces every arithmetic operator with another arithmetic operator;
- LCR replaces every logical connector with another logical connector;
- ROR replaces every relational operator with another relational operator;
- ICR replaces every integer constant c with a different value from the set $\{-1, 0, 1, c-1, c+1\}$.

A similar set of mutation operators is often used in mutation tools for the C language, e.g., by Andrews et al. [3] or Jia et al. [16]. (The Milu tool by Jia et al. [16] can also perform higher-order mutation, which applies multiple syntactic changes to generate one mutant, but we focus our evaluation on the traditional first-order mutation, which applies only one syntactic change to generate each mutant.)

For our evaluation, we used programs from *Coreutils* [10]. *Coreutils* are the basic command-line utilities used in Unix, e.g., `mkdir`, `mv`, or `rm`. *Coreutils* are frequently used as experimental objects in studies involving C programs or LLVM [6], [8], [18], [21]. The source distribution of *Coreutils* includes not only the source code for multiple programs but also several regression tests written for these programs. These tests are often in the form of shell scripts that run the program for various inputs (such as command-line arguments and input files) and check the outputs.

We compiled *Coreutils* using two opposite optimization levels: `-O0` is the basic level that aims at fast compilation and only applies minimal optimizations, and `-O3` is one of the most aggressive optimization levels that applies advanced compiler optimizations. We also mutated each program using each of our mutation operators, applying the compiler optimizations *both* before and after mutation. We identified equivalent and duplicated mutants by comparing the resulting binaries as done by Papadakis et al. [29]. We determined the mutation score by running the mutants against the companion test suites for their programs. For evaluation, we compared the resulting number of mutants and the mutation scores of the test suites across the two levels of compiler optimizations.

Our findings give the following answers to our questions:

- The total number of generated mutants is higher (11.7% overall) at the `-O3` level than at the `-O0` level. This is surprising because the overall number of instructions is lower at the `-O3` level than at the `-O0` level.
- The percentage of equivalent and duplicated mutants is higher (15.9pp[1] on average) at the `-O3` level than at the `-O0` level. This is expected and matches prior work [29], because `-O3` applies more optimizations after applying the mutation operators. Surprisingly, after removing equivalent and duplicated mutants, the number of remaining (non-equivalent, non-duplicated) mutants is *lower* at the `-O3` level than at the `-O0` level. This points to the importance of properly controlling for equivalent and duplicated mutants, especially at high optimization levels. The average percentages of equivalent and duplicated mutants are 7.2% and 13.2%, respectively, at the `-O3` level. (Compared to the reported averages for the C language [29], the percentage for LLVM is similar for equivalent mutants—7.2% vs. 7%—but lower for duplicated mutants—13.2% vs. 21%.)
- The mutation score is persistently lower at the `-O3` level than at the `-O0` level, even when removing equivalent and duplicated mutants. This points to the importance of properly using the mutation score to interpret the results of mutation testing, especially at high optimization levels.
- The results are fairly similar across different mutation operators, indicating that the general conclusions are more likely due to the compiler optimizations than due to the specific operators.

In brief, our study shows that it is promising to apply mutation testing for LLVM with a very high optimization level, but it is necessary to properly control for equivalent and duplicated mutants and to carefully interpret the overall mutation score.

II. Illustrative Overview

We use `cut` to illustrate our evaluation on a concrete example and to introduce some background material. The `cut` program is a standard Unix command-line utility that selects columns or fields from the input and writes them to the standard output. The *Coreutils* source distribution comes with the source code (`src/cut.c`) and 65 tests specifically written for this program. Each test runs `cut` for some given input (provided as the command-line arguments and the content of an input file), and checks that the actual output (in terms of both the content of `stdout` and the return exit code of the program run) matches the expected output. For example, one test gives as input the command-line argument "`-c4`" (to select the 4th character) and the input file with the content "`123`" (with no newline byte before the end of the file). The expected result is an empty string with a new line appended, conforming with `cut`'s spec. All 65 tests can be run with one shell script.

[1]The unit *pp*, from "percentage point", represents the difference of values that are already expressed in percentages.

Listing 1: Unoptimized bitcode (-O0 level)

```
if.end:
  %14 = load i32, i32* %status.addr, align 4
  call void @exit(i32 %14) #8
  unreachable
```

Listing 2: Optimized bitcode (-O3 level)

```
if.end:
  tail call void @exit(i32 %status) #13
  unreachable
```

Fig. 1: Example from `cut` showing instruction count reduction

We want to evaluate the effects of compiler optimizations on mutation testing for `cut`.

A. Clang and LLVM

Our mutation tool-set first uses the Clang [22] compiler front-end to translate `src/cut.c` into LLVM intermediate representation (IR), known as *bitcode*. LLVM provides a rich API for alterations of the bitcode and a large set of compiler optimizations. We modify the `Makefile` build configuration for *Coreutils* such that the compiler produces LLVM bitcode files using a specific optimization level, `-O0` or `-O3`. Note that this step applies optimizations *before* applying the mutations (and a later step will additionally apply optimizations *after* applying the mutations).

B. Compiler Optimizations

Compiler optimizations are semantics-preserving transformations applied to a program with the intention of improving the program's performance. Each optimization is generally intended to make the program smaller or faster. The optimizations are typically applied together to synergistically combine the benefits of each optimization to provide superior performance. While the `-O0` level aims for fast compilation and applies almost no optimizations, the `-O3` level aims to produce very efficient code and applies a large number of optimizations [23]. Specifically, for the `cut` program, the `-O0` level produces LLVM bitcode with 1274 instructions, while the `-O3` level produces 1110 instructions.

For example, Fig. 1 shows a snippet of the `cut` bitcode before and after applying optimizations. It shows the basic block in the `usage` function that returns the status code and then exits. In the unoptimized bitcode, a load instruction retrieves the status code before using it as an argument to the call to `exit`. However, in the optimized bitcode, the tail-call optimization [24] replaces the original call instruction with a tail call, allowing to remove the load instruction, thereby reducing the overall number of instructions and improving the program performance. For `cut`, the total number of call instructions at `-O0` level is 133, of which none is tail. At the `-O3` level, the total number of call instructions is 120, of which 108 are tail calls. Also, the number of load instructions is reduced from 330 at `-O0` to 166 at `-O3`.

C. Mutant Generation

We implemented the mutation operators as manipulations of the LLVM IR. Specifically, we implemented two LLVM passes. The first pass takes as input an LLVM bitcode file and some class of mutation operators (e.g., AOR or ICR). It then finds all bitcode instructions that can be mutated (for AOR, all LLVM instructions that use arithmetic operators such as `add`; and for ICR, all LLVM instructions that have an integer constant), and outputs a set of possible mutations (e.g., replacing a specific `add` instruction with `sub`, `mul`, and `div`; or replacing an integer constant with another integer value). The second pass takes as input an LLVM bitcode file and a specific mutation to apply (as computed by the first pass) and outputs a modified LLVM bitcode file with that mutation applied. Our tool-set invokes the second pass for each and every mutation found by the first pass. Each mutated LLVM bitcode file is then compiled (and linked) into an actual executable, using the LLVM back-end at the same optimization level, either `-O0` or `-O3`, that was used initially by Clang, thus applying optimizations also *after* applying the mutations.

D. Number of Generated Mutants

For `cut`, we obtained a total of 1958 mutants at the `-O0` level and 2547 mutants at the `-O3` level. There were more mutants at the `-O3` level although it had fewer instructions overall than the `-O0` level (1110 vs. 1274), because the `-O3` level had more mutation opportunities; we define a mutation opportunity as a part of an instruction that can be mutated, e.g., the opcode or one of the operands. Across the various classes of mutation operators, we obtained the following numbers of mutants: 100 AOR, 14 LCR, 864 ROR, and 980 ICR at the `-O0` level; and 116 AOR, 102 LCR, 1215 ROR, and 1114 ICR at the `-O3` level.

E. Mutation Score

We next ran the `cut` test suite on each of the mutants, accounting for cases of "rogue" mutants that could affect the entire testing and experimental process. Such cases include mutants that encounter infinite loops (and could block all experiments) or mutants that excessively write to disk. Our tool-set includes a sophisticated runner to handle these cases. If a mutant causes any test in the suite to fail, the mutant is *killed*. Higher-quality test suites kill more mutants, and the percentage of mutants killed is called the *mutation score*. The actual value depends on the generated mutants, which in turn depend on the compiler optimization level. Specifically, for `cut`, we find that the test suite kills a total of 1091 and 1402 of the mutants generated at the `-O0` and `-O3` levels, respectively. The corresponding mutation scores are 55.7% and 55.0%, respectively. The same test suite thus appears seemingly better when evaluated with the mutants generated at the `-O0` level than at the `-O3` level.

F. Equivalent and Duplicated Mutants

Some of the mutants that are generated, while syntactically different in the mutated LLVM, may end up being semantically

```
%12 = and i8 %dash_found.0.ph430.i, 1
%tobool134.i = icmp eq i8 %12, 0
br i1 %tobool134.i, label %L1, label %L2
```

Fig. 2: Example from `cut` for duplicated mutants (`-O3` level)

equivalent to the original `cut` program. No test can kill any equivalent mutant, so ideally all equivalent mutants should be removed from the set of generated mutants. However, determining mutant equivalence is undecidable in general [2], [5]. Our tool-set uses the recently proposed trivial compiler equivalence [29] to perform a bit-by-bit equality comparison between the compiled binaries for the original code and the mutants. If a mutant binary is exactly the same as the original binary, then the mutant is definitely equivalent; if the binaries differ, then we cannot be sure. In the case of `cut`, this technique finds 66 and 111 equivalent mutants at the `-O0` and `-O3` levels, respectively.

Moreover, even if we cannot establish that some mutants are definitely equivalent to the original code, we can find that these mutants are equivalent to one another—following Papadakis et al. [29], we call such mutants *duplicated*. We use the same technique that compares compiled binaries of mutants to find the mutants that are definitely duplicated but not equivalent to the original code. For `cut`, this technique finds 11 duplicated mutants (0.5% of all generated mutants) at the `-O0` level and 360 (14.1% of all generated mutants) at the `-O3` level.

Fig. 2 shows a snippet from `cut` that leads to duplicated mutants. The second instruction compares whether the boolean from the register `%12` is equal to zero and saves the result in the register `%tobool134.i` which is checked in the next branch instruction. The comparison instruction presents two mutation opportunities: one for ROR (replacing the relational operator `eq` with one of {`ne,ugt,uge,ult,ule,sgt,sge,slt,sle`}) and the other for ICR (replacing `0` with one of {`1,-1`}). Four duplicated mutants are generated from this instruction: replacing `eq` with one of {`ne,ugt,sgt`} or replacing `0` with `1`. (Note that these four mutants are *not* equivalent to the original code but are equivalent to one another.) Basically, checking whether a boolean is not equal or greater than `0` (i.e., the boolean is not false) is semantically the same as checking whether the boolean is equal to `1` (i.e., the boolean is true).

Identifying equivalent and duplicated mutants allows us to remove some mutants from mutation testing, which makes mutation testing faster (because we need not run tests on those removed mutants) and provides a more accurate mutation score (because we can use a more precise number of generated and killed mutants). More specifically, we should remove all equivalent mutants, and from each equivalence class of duplicated mutants, we should remove all mutants but one to act as a representative of the equivalence class. We call the set of mutants resulting from this removal the *non-equivalent, non-duplicated (NEND) mutants*. Applying this to `cut`, we end up with 1881 and 2076 NEND mutants at the `-O0` and `-O3` levels, respectively.

G. Revisiting Mutation Score

Revisiting mutation score when considering only NEND mutants, it turns out that the `cut` test suite kills 1085 and 1148 NEND mutants at the `-O0` and `-O3` levels, respectively. The absolute numbers of mutants killed among NEND mutants are lower than the absolute numbers of mutants killed among all generated mutants. The reason is that some equivalence classes of duplicated mutants were killed, and thus removing those duplicated mutants also reduces the number of killed mutants. (Note that removing equivalent mutants never reduces the number of killed mutants, because equivalent mutants cannot be killed.) As a result, we find that the mutation scores are 57.6% and 55.3% for the NEND mutants at the `-O0` and `-O3` levels, respectively. Both of these values are higher than the corresponding values for all generated mutants. We conclude that using only NEND mutants gives a more accurate evaluation of the test suite; users should prefer the `-O3` level, but they should carefully interpret the mutation score obtained at the `-O3` level. We will see that most relationships we have mentioned in this section are not specific to `cut` but actually hold for (almost) all 18 *Coreutils* programs that we evaluate.

III. Experimental Setup

We describe the programs we use in our evaluation, the mutation tool-set we built for the evaluation, and the comparison strategy we used to identify equivalent and duplicated mutants.

A. Object Programs

Our evaluation uses programs from *Coreutils*, a well-studied set of programs frequently used as benchmarks for research in testing [6], [8], [18], [21]. Specifically, we use *Coreutils* version 6.11; while not the most recent, this version is often used in research, including studies on compiler optimizations [8]. We selected 18 programs for our evaluation. We focused on the programs with test directories that explicitly label these programs as test targets (to avoid accidentally killing mutants by tests that do not target the specific program). Out of 27 such programs, our infrastructure had problems with 9, e.g., they had tests with non-deterministic results (known as *flaky tests* [20]). In the case of a flaky test, the output is dependent on the testing environment in addition to the test inputs, so both the original code and its equivalent mutants can pass or fail regardless of the chosen test. We made certain that for all 18 selected programs, (1) all tests pass on the original code, (2) all equivalent mutants produce the same output as the original code, and (3) all duplicated mutants from the same equivalence class return the same result. Each of these 18 programs comes with a number of tests, typically one or more shell scripts that invoke the program multiple times.

B. Compiler Optimizations

In our experiments, we use LLVM 3.8.1, the latest stable version. We selected two opposite optimization levels for our experiments. The `-O0` level provides fast compilation and serves as the baseline for comparison. The `-O3` level is one of highest optimization levels in LLVM 3.8.1, enabling some of the most time-intensive optimizations.

Program	`-O0`						`-O3`					
	Total	Mutation Opportunities					Total	Mutation Opportunities				
	Inst	AOR	LCR	ROR	ICR	Sum	Inst	AOR	LCR	ROR	ICR	Sum
chmod	675	7	15	38	256	316	403	8	15	36	227	286
chown	292	3	1	16	126	146	233	3	3	16	110	132
cut	1274	25	7	96	357	485	1110	29	51	135	417	632
dd	2436	93	74	196	684	1047	2080	91	100	234	705	1130
du	1199	11	6	62	470	549	835	22	15	76	389	502
head	1744	59	7	100	604	770	994	55	17	106	477	655
join	2088	50	6	117	791	964	1958	71	33	208	749	1061
mkdir	251	1	7	12	101	121	159	1	7	9	53	70
mv	604	5	4	31	257	297	372	3	9	28	216	256
readlink	140	1	0	6	44	51	95	1	0	5	35	41
rm	322	2	1	16	162	181	200	2	4	17	113	136
rmdir	349	2	1	21	100	124	199	2	6	23	71	102
tac	871	31	1	54	205	291	581	26	8	60	180	274
tail	3030	74	27	192	1126	1419	2175	87	47	262	997	1393
test	1895	64	24	144	529	761	1711	102	43	237	642	1024
touch	606	5	9	45	242	301	413	5	16	41	219	281
tr	3116	107	25	144	1108	1384	2219	99	104	205	854	1262
wc	1172	43	20	74	346	483	842	47	28	73	319	467
Overall	22064	583	235	1364	7508	9690	16579	654	506	1771	6773	9704

TABLE I: Total number of LLVM instructions and the number of mutation opportunities (per operator class and total), at both `-O0` and `-O3` levels

C. Mutation Tool-Set

We implemented both mutant generation and mutant execution. For mutant generation, we wrote LLVM passes that first identify the points where mutation operators could be applied and then systematically apply these operators to modify the LLVM bitcode files (as described in Section II). We implemented four classes of mutation operators: AOR, LCR, ROR, and ICR (described briefly in Section I). Note that some of the mutation operators for the source language do not apply at the LLVM level. For example, "replace an arithmetic-assignment operator by another operator" replaces C-level assignment operators "`+=`", "`-=`", "`*=`", and "`/=`" with one another, but such assignment operators are de-sugared at the LLVM level. Each of our mutation operators is applied to generate syntactically distinct mutants. We integrated our LLVM passes into the `Makefile` build configuration such that it can generate all the mutants at the specified optimization level. Each of the 18 programs was compiled using each of the two selected optimization levels, producing two original LLVM bitcode files of each program. Each LLVM bitcode file was then subjected to all the mutation operators, generating our set of mutants.

For mutant execution, we created a framework that sandboxes and parallelizes runs of each program's companion test suite against its set of mutants. The framework automatically determines which mutants are killed. When a test terminates, it is easy to determine if a mutant is killed (test failed) or not (test passed). However, in some cases, mutants can exhibit unexpected behavior. First, mutations can massively increase the number of iterations of a loop by altering the guard condition, to the point where mutants can have infinite loops. Our framework handles such cases by time-limiting each test to 30 seconds; mutants that ran out of time are considered killed. Second, a mutant could write arbitrary data to the file system (and *Coreutils* programs and their tests already perform many file-system operations, which makes this case harder to detect). Our framework handles these cases by limiting the size of the files that a process could write. Finally, the entire mutation testing process was very time intensive, as dozens or hundreds of tests needed to be run on hundreds or thousands of mutants per program, so our framework parallelized these runs. We ran our experiments on three 24-core machines with Scientific Linux. When all the tests completed, we were able to assign each test suite a mutation score for each relevant set of mutants.

D. Mutant Comparison

It is important to remove equivalent and duplicated mutants, because they can artificially inflate or deflate the mutation score. We compared the mutants by computing checksums, specifically using `md5sum`, of the final binaries. We then compared the checksums for each mutant to those of the progenitor programs to identify which files have the same content; collisions are highly unlikely using `md5sum`.

IV. Experimental Results

We next discuss the results obtained in our experiments. Table I shows some statistics for the 18 *Coreutils* programs, specifically the total number of the LLVM bitcode instructions and the number of mutation opportunities for various mutation operators at different compiler optimization levels. The number of mutation opportunities is equal to the number of instructions mutated by the corresponding operator for all operators except for ICR, where it reflects the number of integer constant occurrences (and one instruction may have more than one integer constant operand). The last row shows the sum of the values for each column. Overall, there are fewer LLVM instructions at the `-O3` level than at the `-O0` level (16579 vs. 22064). This is expected, as the unoptimized

Program	-O0						-O3					
	#M	#E	E%	#D	D%	#NEND	#M	#E	E%	#D	D%	#NEND
chmod	1069	41	3.8	9	0.8	1019	952	90	9.4	131	13.7	731
chown	467	15	3.2	0	0.0	452	453	67	14.7	41	9.0	345
cut	1958	66	3.3	11	0.5	1881	2547	111	4.3	360	14.1	2076
dd	4208	131	3.1	22	0.5	4055	4721	297	6.2	797	16.8	3627
du	1723	74	4.2	11	0.6	1638	1682	146	8.6	178	10.5	1358
head	2699	110	4.0	27	1.0	2562	2513	250	9.9	306	12.1	1957
join	2902	112	3.8	24	0.8	2766	3980	340	8.5	496	12.4	3144
mkdir	368	23	6.2	3	0.8	342	253	15	5.9	15	5.9	223
mv	907	32	3.5	5	0.5	870	792	82	10.3	98	12.3	612
readlink	192	7	3.6	0	0.0	185	140	5	3.5	12	8.5	123
rm	543	12	2.2	0	0.0	531	458	28	6.1	42	9.1	388
rmdir	479	25	5.2	11	2.3	443	417	16	3.8	48	11.5	353
tac	1151	58	5.0	12	1.0	1081	1120	67	5.9	111	9.9	942
tail	4673	158	3.3	35	0.7	4480	5409	501	9.2	677	12.5	4231
test	3077	75	2.4	66	2.1	2936	4717	248	5.2	710	15.0	3759
touch	1083	52	4.8	17	1.5	1014	983	125	12.7	101	10.2	757
tr	4280	161	3.7	44	1.0	4075	4624	212	4.5	610	13.1	3802
wc	1780	68	3.8	18	1.0	1694	1729	108	6.2	237	13.7	1384
Overall	33559	1220	3.6	315	0.9	32024	37490	2708	7.2	4970	13.2	29812

TABLE II: The number of generated mutants (#M), the number (#E) and percentage (E%) of equivalent mutants, the number (#D) and percentage (D%) of duplicated mutants, and the number of NEND mutants, at both `-O0` and `-O3` levels

code often simply moves data using instructions like `alloca`, `load`, and `store`, while the optimized code removes such instructions (e.g., Fig. 1 discussed in Section II-B). However, there is overall a *similar number* of mutation opportunities at the `-O3` and `-O0` levels (9704 vs. 9690). The optimized and unoptimized versions of the code have a similar number of instructions that perform the actual computations (or operate on constant values) and to which our operators thus apply.

A. Number of Mutants

Table II shows the total number of mutants generated for each program, the number and percentage of equivalent and duplicated mutants, and the number of NEND mutants at both the `-O0` and `-O3` levels. The last row shows the overall values, which are (1) the sums of the numbers of respective mutants in a given column and (2) the overall percentages of equivalent and duplicated mutants (computed as the weighted average across all programs).

From Table II, we see that the overall number of all generated mutants is 11.7% higher at the `-O3` level (37490 mutants) than at the `-O0` level (33559 mutants). However, this relationship between the number of mutants does *not* follow for most of the individual programs. In fact, the relationship is the opposite for all programs except for six (`cut`, `dd`, `join`, `tail`, `test`, and `tr`); these six programs generate a far larger number of mutants at the `-O3` level, thus raising the average and leading to the overall conclusion. The Wilcoxon paired rank test for the numbers of all generated mutants has a p-value of 0.76, indicating that the difference between `-O0` and `-O3` levels is *not* statistically significant.

In brief, we obtain the following answer for RQ1: *The overall number of generated mutants is lower at the `-O0` level than at the `-O3` level, but the opposite holds for most programs and difference is not statistically significant.*

B. Equivalent and Duplicated Mutants

We next analyze the number of equivalent and duplicated mutants in more detail. Table II shows a detailed breakdown for such mutants at both optimization levels. We can see that the overall percentages of both equivalent and duplicated mutants are higher at the `-O3` level (7.2% and 13.2%, respectively) than at the `-O0` level (3.6% and 0.9%, respectively). The comparison between these percentages is similar overall as for almost every individual program.

The overall number of NEND mutants is 6.9% *lower* at the `-O3` level (29812 mutants) than at the `-O0` level (32024 mutants). Moreover, the number of NEND mutants is lower at the `-O3` level than at the `-O0` level for almost every program. Only three programs (`cut`, `join`, and `test`) have more NEND mutants at the `-O3` level than at the `-O0` level. The Wilcoxon paired rank test shows difference between the numbers of NEND mutants ($p < 0.05$). This suggests that mutation testing can be faster at the `-O3` level than at the `-O0` level because there are fewer NEND mutants to run at the `-O3` level, and the more optimized programs likely run faster as well.

In brief, we obtain the following answer for RQ2: *The relative number of both equivalent and duplicated mutants is higher at the `-O3` level than at the `-O0` level; as a result, the overall absolute number of NEND mutants is lower at the `-O3` level than at the `-O0` level (despite the overall absolute number of all generated mutants being higher at the `-O3` level than at the `-O0` level).*

C. Mutation Score

We next consider the mutation score, arguably the most important metric in mutation testing. The number of mutants is an important internal metric because it determines the time needed to perform mutation testing, but the mutation score is an external metric used to compare the quality of test suites. Table III shows the mutation score values. We note two interesting comparisons.

Program	-O0		-O3	
	All	NEND	All	NEND
chmod	35.7	36.9	33.7	35.7
chown	32.7	33.8	29.8	33.3
cut	55.7	57.6	55.0	55.3
dd	32.8	33.9	31.4	31.5
du	41.6	43.4	36.2	38.0
head	17.8	18.7	15.7	17.6
join	54.5	56.6	40.0	42.1
mkdir	52.9	56.4	55.3	57.8
mv	53.8	55.5	50.6	51.4
readlink	39.5	41.0	40.7	39.8
rm	42.7	43.6	35.3	37.1
rmdir	29.4	31.3	28.3	29.7
tac	41.3	43.5	37.0	38.4
tail	31.3	32.4	24.5	26.5
test	37.6	38.7	37.3	38.8
touch	39.4	41.6	38.2	41.0
tr	59.2	61.5	59.2	59.8
wc	32.4	33.8	26.7	26.0
Overall	40.4	41.9	37.0	38.5

TABLE III: The mutation score for all generated mutants and for only NEND mutants, at both `-O0` and `-O3` levels

First, *between the optimization levels*, the corresponding overall mutation score values are lower at the `-O3` level than at the `-O0` level, both for all mutants and for only NEND mutants. Moreover, this holds not only for the overall values but for *most* individual programs, again not only for all mutants but also for only NEND mutants. The Wilcoxon paired rank test shows differences between the mutation score values at `-O0` and `-O3` levels (p < 0.005 for all mutants, and p < 0.001 for only NEND mutants).

In brief, we obtain the following answer for RQ3: *The mutation score values are lower at the `-O3` level than at the `-O0` level both for all mutants and for only NEND mutants.*

Second, *between all mutants and only NEND mutants*, the mutation score value for almost every program is lower for all mutants than the corresponding mutation score value for only NEND mutants (Table III). For example, consider `dd`: at `-O0`, the mutation score value for all mutants (32.8%) is lower than the value for only NEND mutants (33.9%); and similarly, at `-O3`, the value for all mutants (31.4%) is lower than the value for only NEND mutants (31.5%). We do not compare here the value at `-O0` for all mutants and at `-O3` for only NEND mutants because those are not corresponding values.

There is no general relationship between the mutation score values for all mutants and only NEND mutants. Consider some mutation score value $\frac{k}{m}$, where m is the number of all generated mutants and k is the number of mutants killed among those m. If we remove (only) e > 0 equivalent mutants, we must get a higher $\frac{k}{m-e}$. If we remove (only) d > 0 duplicated mutants, and none of them are killed, we must get a higher $\frac{k}{m-d}$. If all the duplicated mutants are killed, we must get a lower $\frac{k-d}{m-d}$. In general, we remove e equivalent and d duplicated mutants, and the number of killed duplicated mutants d' is between 0 and d, so the resulting $\frac{k-d'}{m-e-d}$ must be between $\frac{k}{m-e-d}$ and $\frac{k-d}{m-e-d}$; the resulting mutation score can be higher or lower than the original mutation score.

D. Analysis Across Mutation Operators

We have discovered several relationships between the numbers of all mutants, equivalent and duplicated mutants, NEND mutants, and mutation score values, compared across different compiler optimization levels. However, the analysis so far has been across the mutants generated by *all* mutation operators. Do these relationships vary when considering mutants of each mutation operator individually? We revisit the initial questions while breaking down the numbers for each operator classes.

Tables IV and V show a detailed breakdown, per operator class, of the left side of Table II and the right side of Table II, respectively. Table VI shows the detailed breakdown per operator class of Table III. When considering the breakdown into individual operator classes, it is important to note how to treat duplicated mutants, because mutants can be duplicated across different operator classes (e.g., a mutant generated using the AOR operator can be a duplicate of a mutant generated using the ICR operator). In these tables, we compute duplicated mutants for each operator as if the only mutants that exist are the mutants generated by that operator. For example, if two mutants M_a and M_i are duplicates of each other, but M_a is generated by the AOR operator and M_i by the ICR operator, when considering mutants from AOR, M_a is counted as NEND and not counted as a duplicated mutant; likewise, when considering mutants from ICR, M_i is counted as NEND. The sum of all the duplicated mutants we report for each operator is lower than the number we report overall in Table II, because many duplicated mutants when considering all operators together may not be duplicated when considering only one operator.

Table VII summarizes the relationships across all operators and per each operator. Most relationships that hold for the mutants generated by all operators together also hold when considering only the mutants generated by each operator individually. One exception concerns the overall number of all generated mutants for the ICR operator: the overall number of all generated mutants at the `-O3` level is *higher* than the overall number of all generated mutants at the `-O0` level for all operators except ICR. Though there are a few programs where the number of generated mutants at `-O3` for ICR is higher than at `-O0` (`cut`, `dd`, `join`, `tail`, and `test`), for the majority of programs this is not the case, and these exceptions do not have many more generated mutants at the `-O3` level. The other exceptions concern the number of NEND mutants for the LCR operator and the ROR operator. Concerning the LCR operator, it does not generate many equivalent nor duplicated mutants at either optimization level. As such, since the LCR operator generates more mutants overall at the `-O3` level, it continues to have more NEND mutants at the `-O3` level as well. Concerning the ROR operator, the difference between the number of NEND mutants at the different optimization levels is relatively small (12265 at `-O0` vs. 12816 at `-O3`), and we see that there are actually only six programs that have more NEND mutants at the `-O3` level than at the `-O0` level (`cut`, `du`, `join`, `tail`, `test`, and `tr`).

Program	`-O0`															
	AOR				LCR				ROR				ICR			
	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND
chmod	28	0.0	0.0	28	30	0.0	0.0	30	342	0.0	0.0	342	669	6.1	1.3	619
chown	12	0.0	0.0	12	2	0.0	0.0	2	144	0.0	0.0	144	309	4.8	0.0	294
cut	100	0.0	0.0	100	14	0.0	0.0	14	864	1.0	0.0	855	980	5.8	1.1	912
dd	370	0.0	0.0	370	148	0.0	0.0	148	1764	0.0	0.0	1764	1926	6.8	1.1	1773
du	44	0.0	0.0	44	12	0.0	0.0	12	558	0.0	0.0	558	1109	6.6	0.9	1024
head	236	0.0	0.0	236	14	0.0	0.0	14	900	0.0	0.0	900	1549	7.1	1.7	1412
join	200	0.0	0.0	200	12	0.0	0.0	12	1053	0.0	0.0	1051	1637	6.7	1.4	1503
mkdir	4	0.0	0.0	4	14	0.0	0.0	14	108	0.0	0.0	108	242	9.5	1.2	216
mv	20	0.0	0.0	20	8	0.0	0.0	8	279	0.0	0.0	279	600	5.3	0.8	563
readlink	4	0.0	0.0	4	0	N/A	N/A	0	54	0.0	0.0	54	134	5.2	0.0	127
rm	8	0.0	0.0	8	2	0.0	0.0	2	144	0.0	0.0	144	389	3.0	0.0	377
rmdir	8	0.0	0.0	8	2	0.0	0.0	2	189	0.0	0.0	189	280	8.9	3.9	244
tac	124	0.0	0.0	124	2	0.0	0.0	2	486	0.0	0.0	486	539	10.7	2.2	469
tail	296	0.0	0.0	296	54	0.0	0.0	54	1728	0.0	0.0	1728	2595	6.0	1.3	2403
test	256	0.0	0.0	256	48	0.0	0.0	48	1296	0.0	0.0	1296	1477	5.0	4.4	1336
touch	20	0.0	0.0	20	18	0.0	0.0	18	405	0.0	0.0	405	640	8.1	2.6	571
tr	428	0.0	0.4	426	50	0.0	0.0	50	1296	0.0	0.0	1296	2506	6.4	1.6	2305
wc	172	0.0	1.1	170	40	0.0	0.0	40	666	0.0	0.0	666	902	7.5	1.5	820
Overall	2330	0.0	0.1	2326	470	0.0	0.0	470	12276	0.0	0.0	12265	18483	6.5	1.6	16968

TABLE IV: The number of generated mutants (#M), the percentages of equivalent (E%) and duplicated (D%) mutants, and the number of NEND mutants, split across mutation operators classes at the `-O0` level

Program	`-O3`															
	AOR				LCR				ROR				ICR			
	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND
chmod	32	0.0	25.0	24	30	0.0	0.0	30	324	8.9	17.2	239	566	10.7	6.3	469
chown	12	0.0	25.0	9	6	0.0	0.0	6	144	6.9	16.6	110	291	19.5	1.3	230
cut	116	0.0	26.7	85	102	0.0	1.9	100	1215	7.5	13.4	960	1114	1.7	3.1	1060
dd	362	0.0	23.2	278	200	7.5	3.0	179	2106	7.3	17.2	1588	2053	6.2	4.8	1825
du	88	0.0	17.0	73	30	0.0	0.0	30	684	5.7	11.7	565	880	12.1	3.7	740
head	220	0.0	12.7	192	34	0.0	0.0	34	954	5.8	12.2	781	1305	14.8	5.5	1038
join	284	2.4	30.9	189	66	0.0	0.0	66	1872	7.2	10.1	1546	1758	11.2	3.4	1500
mkdir	4	0.0	0.0	4	14	0.0	0.0	14	81	9.8	8.6	66	154	4.5	1.9	144
mv	12	0.0	25.0	9	18	0.0	5.5	17	252	5.1	19.0	191	510	13.5	4.3	419
readlink	4	0.0	50.0	2	0	N/A	N/A	0	45	6.6	11.1	37	91	2.2	3.3	86
rm	8	0.0	25.0	6	8	0.0	0.0	8	153	5.2	15.6	121	289	6.9	1.7	264
rmdir	8	0.0	37.5	5	12	0.0	0.0	12	207	5.8	8.7	177	190	2.1	8.9	169
tac	104	0.0	5.7	98	16	12.5	0.0	14	540	5.5	10.5	453	460	7.6	4.5	404
tail	348	0.0	12.6	304	94	0.0	1.0	93	2358	5.8	13.0	1912	2609	13.9	5.2	2109
test	408	0.0	19.1	330	86	0.0	0.0	86	2133	5.7	10.2	1792	2090	5.9	9.9	1758
touch	20	0.0	30.0	14	32	0.0	0.0	32	369	7.5	11.6	298	562	17.2	3.9	443
tr	396	0.0	24.4	299	208	0.4	0.0	207	1845	5.5	15.4	1457	2175	4.9	2.4	2014
wc	188	0.0	17.5	155	56	0.0	1.7	55	657	6.0	14.3	523	828	8.2	4.5	722
Overall	2614	0.2	20.3	2076	1012	1.7	1.0	983	15939	6.4	13.1	12816	17925	9.2	4.8	15394

TABLE V: The number of generated mutants (#M), the percentages of equivalent (E%) and duplicated (D%) mutants, and the number of NEND mutants, split across mutation operators classes at the `-O3` level

In brief, we obtain the following answer for RQ4: *The effects of `-O0` and `-O3` levels on mutation testing are most likely due to compiler optimizations and not due to specific mutation operators.*

E. Visual Summary

As an overall visual summary of all our high-level results, Fig. 3 compares the number of instructions, mutation opportunities, generated mutants (both all and NEND), and mutation score (both all and NEND) between optimization levels `-O0`, shown in (dark) blue, and `-O3`, shown in (light) red. To summarize, although there are fewer instructions in the programs compiled at the `-O3` level than at the `-O0` level, there are actually slightly more mutation opportunities at the `-O3` level, which in turn leads to more mutants generated. However, when we keep only NEND mutants, there are fewer NEND mutants at the `-O3` level. For both all mutants and only NEND mutants, the mutation score values are lower at the `-O3` level.

V. Threats to Validity

Internal. Our implementation of mutation testing and the scripts for running the experiments may contain bugs. To reduce the risks, we used the well-known framework LLVM and reviewed our code and scripts to check basic functionality. We built our own scripts to run the experiments, collect results, and analyze them. We performed sanity checks on the numbers that the scripts generated, e.g., we checked that equivalent and duplicated mutants give the same results, as they should (Section III-A). We also manually inspected some outliers to confirm that the results were correct.

Program	-O0								-O3							
	AOR		LCR		ROR		ICR		AOR		LCR		ROR		ICR	
	All	NEND	All	NEND	All	NEND	All	NEND	All	NEND	All	NEND	All	NEND	All	NEND
chmod	39.2	39.2	66.6	66.6	36.8	36.8	33.6	35.3	53.1	54.1	73.3	73.3	36.1	37.6	29.1	32.6
chown	50.0	50.0	50.0	50.0	31.9	31.9	32.3	34.0	50.0	44.4	50.0	50.0	35.4	35.4	25.7	32.1
cut	70.0	70.0	57.1	57.1	59.4	60.1	50.9	54.0	68.1	68.2	72.5	72.0	54.3	55.6	52.8	54.0
dd	35.4	35.4	48.6	48.6	35.5	35.5	28.7	30.7	33.9	29.8	38.0	39.1	36.3	37.4	25.2	27.2
du	72.7	72.7	58.3	58.3	46.5	46.5	37.6	40.2	61.3	56.1	73.3	73.3	34.0	34.8	34.0	38.7
head	14.4	14.4	7.1	7.1	21.2	21.2	16.5	17.9	13.1	11.4	14.7	14.7	16.3	17.8	15.7	18.7
join	65.5	65.5	100.0	100.0	54.4	54.5	53.0	56.6	44.7	45.5	46.9	46.9	40.3	41.9	38.7	42.6
mkdir	100.0	100.0	71.4	71.4	51.8	51.8	51.6	56.9	100.0	100.0	71.4	71.4	54.3	57.5	53.2	55.5
mv	100.0	100.0	87.5	87.5	59.5	59.5	49.1	51.5	100.0	100.0	83.3	82.3	63.1	62.3	42.1	46.3
readlink	100.0	100.0	N/A	N/A	57.4	57.4	30.6	32.2	100.0	100.0	N/A	N/A	53.3	54.0	31.8	33.7
rm	50.0	50.0	0.0	0.0	37.5	37.5	44.7	46.1	50.0	66.6	50.0	50.0	33.3	34.7	35.6	37.5
rmdir	50.0	50.0	50.0	50.0	32.8	32.8	26.4	29.5	50.0	40.0	50.0	50.0	24.6	27.1	30.0	30.7
tac	46.7	46.7	50.0	50.0	46.5	46.5	35.4	39.6	66.3	66.3	31.2	35.7	38.5	38.8	28.9	31.1
tail	53.7	53.7	11.1	11.1	30.7	30.7	29.6	31.5	45.6	46.3	23.4	22.5	24.2	24.0	22.0	25.7
test	38.2	38.2	45.8	45.8	41.2	41.2	34.0	36.1	40.6	40.3	41.8	41.8	33.4	34.7	40.4	43.3
touch	60.0	60.0	11.1	11.1	46.1	46.1	35.3	38.7	60.0	57.1	31.2	31.2	44.1	44.9	33.9	39.5
tr	80.1	80.0	52.0	52.0	60.1	60.1	55.3	59.0	85.3	83.6	37.9	38.1	59.0	60.1	56.7	59.3
wc	44.7	45.2	40.0	40.0	28.3	28.3	32.8	35.6	42.0	40.0	41.0	40.0	21.9	21.4	26.0	26.8
Overall	51.4	51.4	45.1	45.1	41.9	42.0	37.8	40.5	49.2	47.5	43.7	43.9	37.3	38.1	34.6	38.1

TABLE VI: The mutation score split across mutation operator classes for all generated mutants and only NEND mutants

Relationship	Operators				
	All	AOR	LCR	ROR	ICR
for all generated mutants, overall #M at `-O3` > #M at `-O0`	yes	yes	yes	yes	no
for only NEND mutants, overall #NEND at `-O3` < #NEND at `-O0`	yes	yes	no	no	yes
overall E% at `-O3` ≥ E% at `-O0`	yes	yes	yes	yes	yes
overall D% at `-O3` ≥ D% at `-O0`	yes	yes	yes	yes	yes
for all generated mutants, overall mutation score at `-O3` < mutation score at `-O0`	yes	yes	yes	yes	yes
for only NEND mutants, overall mutation score at `-O3` < mutation score at `-O0`	yes	yes	yes	yes	yes

TABLE VII: Relationships for mutants generated by all operators together vs. mutants generated by each operator individually

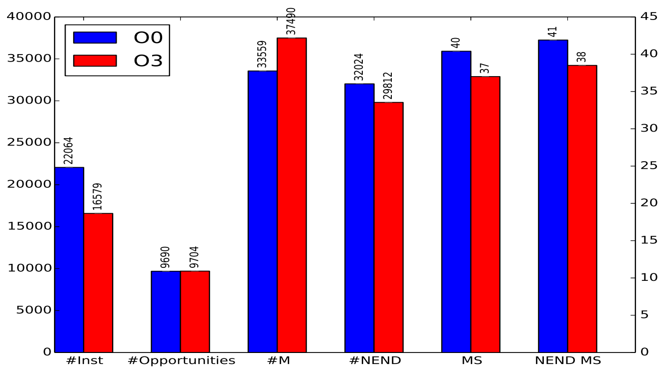

Fig. 3: The overall number of instructions, mutation opportunities, generated mutants (all and NEND), and mutation score (all and NEND)

External. The programs and tests that we used for our empirical study are a subset of all available software and may not be representative. Thus, our findings may not generalize to all software. To address this threat, we selected a total of 18 open-source programs from the widely used *Coreutils* distribution. We used all of the regression tests that come with each *Coreutils* program and specifically target it.

Construct. We use the technique proposed by Papadakis et al. to identify equivalent and duplicated mutants [29]. The technique is a heuristic and provides only a lower bound on the number of such mutants; we cannot claim we identified *all* equivalent and duplicated mutants. We also chose the basic `-O0` optimization level in LLVM as our baseline and compared it with one of the most advanced levels, `-O3`. The results could differ for other combinations of compiler optimizations, but we hypothesize that the general result holds: using higher optimization levels is beneficial, as long as equivalent and duplicated mutants are removed, and the mutation score is properly interpreted.

VI. Related Work

Mutation testing was introduced almost four decades ago [7], [12]. Since its introduction, mutation testing has been used for a number of different programming languages and in numerous applications; Jia and Harman present a recent survey [17]. Mutation testing has gained popularity in testing research primarily for its ability to assess the quality of test suites. Substantial progress has been made toward turning mutation testing into a broadly applicable and fully automated approach; several tools have been created explicitly for this purpose. The most related to our work are two tools that operate on the LLVM IR, namely Sen and Sousa's testing framework for automated mutant generation for transaction level modeling [35] and Schulte's `llvm-mutate` tool (which uses a different set of mutation operators than what we use) [33], [34]. However, their previous work did not study the effect of the LLVM's compiler optimizations on mutation testing (including the number of generated mutants, duplicated and equivalent mutants, and the mutation score).

Despite the progress made on increasing the applicability of mutation testing, the approach still suffers from a number of issues. One key issue is the *mutant equivalence problem*, i.e., determining which mutants are semantically equivalent to the original program. (There are variations in the definition based on the scope of testing, e.g., Ellims et al. suggested a "resource-aware" view of mutants that encompasses memory and time usage as well as functional output [9].) Budd and Angluin first noted that determining mutant equivalence automatically is generally undecidable [5]. However, a number of heuristics have been developed for identifying equivalent and duplicated mutants in some cases.

The foundational work was done by Baldwin and Sayward in their study on the use of compiler optimizations to determine mutant equivalencies [4]. Using established compiler optimization techniques, they attempted to identify equivalent mutants by either "optimizing" or "de-optimizing" the produced programs and comparing them to the original. Offutt and Pan created a new approach to the problem by formulating the question of equivalence as a constraint satisfaction problem [27], [28]. In their approach, constraints are generated through analysis of the mutant's path conditions, and empirical evaluations showed that this approach tended to be more powerful than the compiler optimization technique [25].

Papadakis et al. presented *trivial compiler equivalence*, a technique that compares a mutant's machine code to that of its progenitor program to determine whether or not it is equivalent [29]. This technique was already broadly used in the field of compiler optimizations to determine where optimizations had yielded no improvement or other change.

In contrast to the technique proposed by Papadakis et al. that aims to find *definitely* equivalent mutants, several studies have proposed heuristics to help identify mutants that are *likely* (non-)equivalent. Schuler et al. proposed two such techniques for ranking mutants based on code coverage and dynamically inferred invariants [31], [32]. Grün et al. defined an impact function characterizing the difference between a mutant's execution and the original program's execution [11]; they find mutants with lower impact were more likely to be equivalent.

In a similar spirit to these heuristics, Harman et al. developed a technique for equivalence detection using program slicing [13], [15]. Their method does not automatically detect equivalent mutants after generation but does reduce their number during generation, and it also assists in the manual process of equivalence analysis by simplifying the program to a minimal state representing at least a partial answer to the equivalence question. Adamopoulos et al. proposed a different approach that uses genetic algorithms wherein mutants are evaluated using a fitness function that has a much lower value for equivalent mutants [1]. Their approach allows mutants to evolve alongside the test suites that support the program while reducing the number of equivalent mutants generated.

Rajan et al. studied the effect of program transformations on code coverage, specifically MC/DC [14], [30]. In their study, Rajan et al. used mutation testing as an enabling method but did so without regard for compiler optimization levels.

To the best of our knowledge, no previous work examined the impact of the compiler optimization level on mutation testing (including the impact on equivalent and duplicated mutants and especially the impact on mutation score) as our study does. However, ours is not the first study to examine the effects of compiler optimizations and other transformations on advanced testing and verification tools. Most recently, Dong et al. investigated the interactions between compiler optimizations and symbolic execution [8]. They found that the same transformations that speed up concrete execution can negatively impact symbolic execution, especially in combination. In contrast, we find that compiler optimizations at the highest level not only can speed up program execution but also can substantially help in mutation testing.

VII. Conclusions

We presented a study of the effects of compiler optimizations, which are widely used semantics-preserving transformations aimed at improving program performance, on mutation testing, a research approach for evaluating the quality of test suites. While mutation testing and compiler optimizations are two well-studied approaches, they are seldom used together. Our study aims to find new opportunities that enhance the effectiveness and application of mutation testing by leveraging modern compiler infrastructures. We target LLVM, a popular compiler infrastructure that supports multiple languages. Our evaluation uses 18 *Coreutils* programs. Some of the findings about the number of mutants and the mutation scores on optimized and unoptimized programs surprised us.

The overall conclusion is that mutation testing can use very high optimization levels, but one should remove equivalent and duplicated mutants, and one should carefully interpret the overall mutation score. Note that our conclusion about the mutation score views mutation testing only as a means to evaluate test suites and does not necessarily aim to provide guidance for how to generate new tests to kill more mutants; indeed, it would be hard for a human to reason about the changes made to the optimized program and successfully construct test inputs that could kill the mutant. We hope that our work encourages others to further integrate mutation testing with compiler infrastructures, and to enhance the efficacy and usage of mutation testing.

Acknowledgements

We thank Ben Lambeth for his help with our experimental scripts; Owolabi Legunsen for his discussions about this project; and Vimuth Fernando, Milica Hadži-Tanović, Sasa Misailovic, Peiyuan Zhao, and the anonymous reviewers for feedback on previous drafts of this paper. This research was partially supported by National Science Foundation Grant Nos. CNS-1239498, CCF-1319688, CCF-1409423, CCF-1421503, CCF-1438982, and CCF-1439957.

References

[1] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *GECCO*, pages 1338–1349, 2004.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, 2005.

[4] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Technical report, Yale University, 1979.

[5] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[8] S. Dong, O. Olivo, L. Zhang, and S. Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *ISSRE*, pages 205–215, 2015.

[9] M. Ellims, D. Ince, and M. Petre. The Csaw C mutation tool: Initial results. In *TAIC PART*, pages 185–192, 2007.

[10] F. S. Foundation. Coreutils – GNU core utilities. http://www.gnu.org/software/coreutils/coreutils.html.

[11] B. J. M. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *ICSTW*, pages 192–199, 2009.

[12] R. G. Hamlet. Testing programs with the aid of a compiler. *TSE*, 3(4):279–290, 1977.

[13] M. Harman, R. Hierons, and S. Danicic. *Mutation Testing for the New Century*. Kluwer Academic Publishers, 2001.

[14] M. P. Heimdahl, M. W. Whalen, A. Rajan, and M. Staats. On MC/DC and implementation structure: An empirical study. In *DASC*, pages 5.B.3–1–5.B.3–13, 2008.

[15] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *STVR*, 9(4):233–262, 1999.

[16] Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *TAIC PART*, pages 94–98, 2008.

[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5):649–678, 2011.

[18] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, pages 193–204, 2012.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004.

[20] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *FSE*, pages 643–653, 2014.

[21] P. D. Marinescu and C. Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE*, pages 716–726, 2012.

[22] U. of Illinois. Clang: A C family language frontend for LLVM. http://clang.llvm.org/.

[23] U. of Illinois. The LLVM compilation infrastructure. http://llvm.org/.

[24] U. of Illinois. Tail call optimization. http://llvm.org/docs/CodeGenerator.html#tail-call-optimization.

[25] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *STVR*, 4(3):131–154, 1994.

[26] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *TOSEM*, 5(2):99–118, 1996.

[27] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *COMPASS*, pages 224–236, 1996.

[28] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *STVR*, 7(3):165–192, 1997.

[29] 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*, pages 936–946, 2015.

[30] A. Rajan, M. W. Whalen, and M. P. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *ICSE*, pages 161–170, 2008.

[31] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *ESEC/FSE*, pages 297–298, 2009.

[32] D. Schuler and A. Zeller. (Un-)covering equivalent mutants. In *ICST*, pages 45–54, 2010.

[33] E. Schulte. llvm-mutate. http://eschulte.github.io/llvm-mutate/.

[34] E. Schulte. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution*. PhD thesis, University of New Mexico, 2014.

[35] M. Sousa and A. Sen. Generation of TLM testbenches using mutation testing. In *CODES+ISSS*, pages 323–332, 2012.