SRCIROR: A Toolset for Mutation Testing of C Source Code and LLVM Intermediate Representation

Farah Hariri
University of Illinois at Urbana-Champaign
Illinois, USA
hariri2@illinois.edu

August Shi
University of Illinois at Urbana-Champaign
Illinois, USA
awshi2@illinois.edu

ABSTRACT

We present *SRCIROR* (pronounced "sorcerer"), a toolset for performing mutation testing at the levels of `C/C++` source code (SRC) and the LLVM compiler intermediate representation (IR). At the SRC level, *SRCIROR* identifies program constructs for mutation by pattern-matching on the Clang AST. At the IR level, *SRCIROR* directly mutates the LLVM IR instructions through LLVM passes. Our implementation enables *SRCIROR* to (1) handle any program that Clang can handle, extending to large programs with a minimal overhead, and (2) have a small percentage of invalid mutants that do not compile. *SRCIROR* enables performing mutation testing using the same classes of mutation operators at both the SRC and IR levels, and it is easily extensible to support more operators. In addition, *SRCIROR* can collect coverage to generate mutants only for covered code elements. Our tool is publicly available on GitHub (https://github.com/TestingResearchIllinois/srciror). We evaluate *SRCIROR* on Coreutils subjects. Our evaluation shows interesting differences between SRC and IR, demonstrating the value of *SRCIROR* in enabling mutation testing research across different levels of code representation.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**;

KEYWORDS

Software Testing, Mutation Testing

ACM Reference Format:
Farah Hariri and August Shi. 2018. SRCIROR: A Toolset for Mutation Testing of C Source Code and LLVM Intermediate Representation. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3238147.3240482

1 INTRODUCTION

Software testing is commonly used in industry for quality assurance. One key challenge of software testing is to properly evaluate the quality of test suites in terms of their bug-finding capability. A test suite with a large number of tests, or that achieves a high statement or branch coverage, does not necessarily have a high bug-finding capability.

Mutation testing is widely used in research to evaluate the quality of test suites [13], and it has recently started to gain momentum in industry as well [17]. Mutation testing proceeds in two steps. The first step is mutant generation. A mutant is a modified version of the original program obtained by applying a mutation operator. A mutation operator is a program transformation that introduces a small syntactic change to the original program. The second step of mutation testing is to run the test suite and determine which mutants are killed. A mutant is killed if the tests behave differently, typically in their pass/fail status, when run on the mutant compared against running the tests on the original program. Mutation testing produces a measure of quality of the test suite called the *mutation score*. The mutation score of a given test suite is the percentage of mutants killed by that test suite out of the total number of generated mutants.

Multiple mutation testing tools were developed that perform mutation at different levels. Traditional mutation testing is performed at the level of source code (*SRC*), e.g., for `C` [4–8, 11, 12], and `Java` [14, 15]. More recently, mutation testing has been applied at the level of compiler intermediate representation (*IR*), e.g., for the LLVM IR [10, 18–20]. However, the currently available tools do not meet all researchers' needs. First, some of the tools apply transformations on the text of the source code without performing any parsing [5, 9]. Such tools generate a large number of mutants that do not compile and can waste a lot of the mutant generation time. They also can miss generating some mutants as they rely on syntactic matching to detect mutation opportunities. Second, some tools implement their own parsing trees and, therefore, may not support all language constructs and would fail to generate mutants on even moderately sized programs. Lastly, there does not exist one framework that supports mutation at different levels (SRC and IR) allowing fair comparison and easy extension for supporting more operators.

We present *SRCIROR* (pronounced "sorcerer"), a toolset for performing mutation testing at the levels of `C/C++` source code (SRC) and the LLVM compiler intermediate representation (IR). At the SRC level, *SRCIROR* identifies program constructs by performing pattern-matching on the Clang AST. *SRCIROR* then applies the relevant mutation operators on the found constructs. At the IR level, *SRCIROR* finds the instructions that should be mutated using an LLVM pass and then directly mutates those IR instructions. Our implementation enables *SRCIROR* to (1) handle any program that Clang can handle, extending to large programs with a minimal overhead, and (2) have a small percentage of invalid mutants that

ASE '18, September 3–7, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240482

do not compile. *SRCIROR* enables performing mutation testing using the same mutation operator classes at both the SRC and IR levels, and *SRCIROR* is easily extensible to support more operators. In addition, *SRCIROR* can collect coverage to generate mutants only for covered code elements. *SRCIROR* is open-source and is publicly available at https://github.com/TestingResearchIllinois/srciror.

We evaluate *SRCIROR* on five subjects from Coreutils. Coreutils is a library of command line utilities for Unix that is widely used in research. We are the first to perform mutation testing on the entire code of our five subjects for both SRC and IR levels. Our evaluation shows interesting results demonstration the value of *SRCIROR* in enabling mutation testing research across different levels.

2 MUTATION TOOLS IMPLEMENTATION

In this section we describe the mutation operators in *SRCIROR*. We then describe in detail the implementation of *SRCIROR*'s SRC and IR mutant generation components. Finally, we describe using code coverage to filter out mutants for mutation testing.

2.1 Mutation Operators

We define four mutation operators in common at both SRC and IR levels. A similar set of mutation operators is often used in the existing mutation tools for the `C` language, e.g., by Andrews et al. [4, 5] or Jia and Harman [11, 12]. These four mutation operators are:

- *AOR* replaces every arithmetic operator from the set {`+, -, *, /, %`} with a *different* arithmetic operator from the same set. At the SRC level, the AOR class also includes replacing the arithmetic assignment operators from the set {`+=, -=, *=, /=, %=`} with other operators of that same set. Replacing arithmetic assignment operators 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 a *different* operator from the same set. 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 a *different* operator from the same set. 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, the operator involves replacing every IR instruction from the set {`eq, ne, ugt, uge, ult, ule, sgt, sge, slt, sle`} with a *different* predicate 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}.

2.2 Source-level Mutant Generation Tool

We implement our source (SRC)-level mutant generation tool as a source-to-source transformation tool based on Clang (version 3.8.1). The architecture of our implementation is shown in Figure 1. We perform the SRC mutation in three steps. First, we use Clang to parse the input files and build an abstract syntax tree (AST). Second, we use *AST Matchers* [2] (combined with *LibTooling* [1]) to search for candidate mutation locations in the AST. Finally, for each of these candidates, we use these same two libraries to modify the Clang AST, performing a source-to-source transformation that mutates the AST based on the mutation, generating a different mutated source file for each mutation.

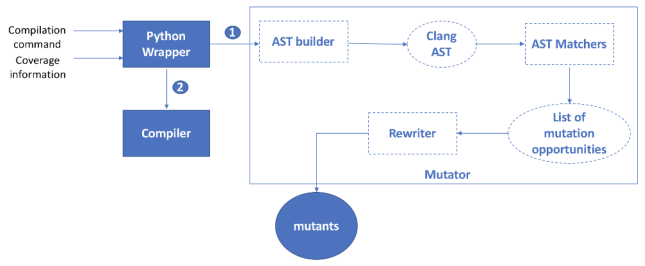

Figure 1: SRC Mutator Architecture

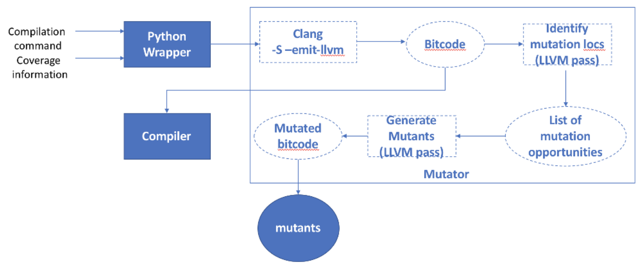

Figure 2: IR Mutator Architecture

SRCIROR supports generating mutants in different files for a project. This feature 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 the Coreutils tools is defined in a utility directory that gets compiled into a shared library `libcoreutils.a` that links to the executable. Failing to generate mutants for code from `libcoreutils.a` decreases the confidence in the value of the mutation testing results.

2.3 IR-level Mutant Generation Tool

For IR-level mutant generation, *SRCIROR* uses transformation passes in the LLVM compiler infrastructure (LLVM version 3.8.1) to generate mutants (Figure 2). First, we use Clang with the flags `-S -emit-llvm` to generate LLVM bitcode files representing the code. Next, we apply two LLVM passes. The first LLVM pass takes in the bitcode file and generates as output the list of *mutation opportunities*; a mutation opportunity is defined by a location (specific LLVM instruction and one of its operands) that can be mutated and the mutation type (the mutation operator and the value to substitute the operand for). The second LLVM pass takes as input a bitcode file and the mutation opportunity to apply, and then actually applies the mutation, creating a new mutated bitcode file for each mutation. Finally, the mutated bitcode is passed along to the compiler to resume the original compilation, resulting in a final compiled mutant. Note that the second step of generating mutants is carried by the user; a simple loop is needed to go over each mutation opportunity generated in the first step and feed it into *SRCIROR* along with the original bitcode file to perform the mutation.

For both SRC and IR, we create Python wrapper scripts that are called by the project build scripts instead of Clang. The wrapper scripts implement the same interface as Clang, performing the same operations as described earlier for SRC and IR mutations, and then delegating the remaining compilation commands back to the actual Clang. For example, in the case of SRC, the Python script calls the mutator on the commands that have source files in them. In the case of IR, the Python script uses the commands along with some inserted flags to first generate LLVM bitcode, call the mutator LLVM passes on the bitcode, and then finish compiling the mutated bitcode by delegating back to the original Clang compiler.

2.4 Incorporating Coverage

While we can run tests against all the mutants generated at the SRC and IR level to compute the mutation score, the tests may not necessarily cover some of the generated mutants. If tests do not cover certain mutants, then those tests cannot kill such mutants. While it is important for a developer using the mutation testing tool to know when some mutants are not even covered, as it indicates a weakness in the test suite, sometimes a developer wants to know just how good the test suite is on the mutants already covered. Furthermore, checking only mutants killed of the mutants covered leads to faster mutation testing as fewer mutants are run. We allow for using code coverage to filter out mutants that should not be run with tests (Figures 1 and 2).

At the SRC-level, we run tests first on code instrumented using `llvm-cov gcov` to collect simple coverage. Then we feed in the covered lines as input to the mutator; the pattern-matching logic will only generate mutants of covered lines.

At the IR-level, there is no existing tool like `llvm-cov gcov` that collects code coverage of IR instructions. As such, we implement our own code coverage tool at the IR level as a new LLVM pass. The pass iterates through each LLVM instruction while keeping a counter, giving a unique count for each instruction. At each instruction, the pass inserts a call to a helper coverage instrumentation method. When the tests are executed on the instrumented code, executing the call records the instruction count, which is then written into a trace file. This trace file represents the coverage at the IR-level. In the first LLVM mutator pass that determines what instructions to mutate, the pass counts the instructions in the same way as in the coverage pass, and then the pass only outputs a mutation for an instruction if its count matches an instruction count from the input coverage information.

3 EVALUATION

3.1 Splitting Coreutils Tests

We perform an evaluation of using *SRCIROR* on programs from Coreutils version 6.11. 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, many of the programs have test script files that contains tests. If we were to execute such a test script for a given program directly on the original and mutated versions of the program, it would execute *all* tests and report a failure if *any* of the tests fails. Therefore, we would just know if a mutant is killed or not, but 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 one were to use a mutation testing tool 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, it is often desirable to obtain the full test-mutant matrix because it can facilitate a further analysis of mutants normally needed in mutation research, e.g., computation of minimal mutant sets [3]. To get the full matrix for the programs, we manually analyzed all the test script files for the Coreutils programs used in our evaluation, and we split each long script into several shorter scripts that each runs an individual test.

We split long test scripts into shorter test scripts through a combination of automated transformations (whenever it was possible) and manual changes. To ensure that our process for splitting the test scripts does not affect the validity of the results, we executed all shorter test scripts on their respective programs to verify that each of them gives the expected result on the original code. More precisely, executing a test on a program in Coreutils can give one of the three possible results: PASS, FAIL, or SKIP. The tests are skipped during execution when their precondition state is not established, which can happen for a number of reasons. One reason that we commonly found for skipped tests was that they required to be run with the root privilege level. Another reason was that a few tests required the presence of more than one disk partition mounted on the file system. These tests report the SKIP result for the original program as well as for any mutant generated for the program. Further, we inspected all tests that were getting skipped after our splitting of long test scripts into shorter test scripts. For most cases, the test was also originally skipped in the longer script due to unavailable privileges or resources, which is the correct behavior. For a few cases, the test started being skipped after our splitting. We carefully inspected the latter cases and found out that some tests were getting skipped because their setup was getting skipped—this setup usually sets some test environment variables and is performed when all tests are run by invoking `make check` from the top-most test directory; our shorter scripts do not invoke tests that way. However, the most important aspect is that the same tests are skipped consistently, and thus they do *not* affect mutation testing analyses using the artifact. The Coreutils artifacts (splitted tests along with the scripts we wrote to perform the splitting) are available at
http://mir.cs.illinois.edu/farah/artifacts/coreutils-artifact.tar.gz.

3.2 Results

Tables 1 and 2 show the numbers of mutants generated by *SRCIROR* that are covered by tests at the SRC and IR levels, respectively (columns "#M"). *SRCIROR* generates many more mutants at the IR level than at the SRC level, 15944 versus 4261, respectively.

Table 1: Number of SRC mutants generated and the number/percentage of them that are equivalent/duplicated

Program	Tests	Overhead	SRC					
			#M	#E	E%	#D	D%	#NEND
factor	31	32.14	364	12	3.3	37	10.2	315
head	85	54.06	946	43	4.5	123	13.0	780
seq	37	72.96	989	40	4.0	123	12.4	826
stat	68	150.96	1619	72	4.4	246	15.2	1301
unexpand	38	18.02	343	8	2.3	37	10.8	298
Overall	259	328.14	4261	175	4.1	566	13.3	3520

Table 2: Number of IR mutants generated and the number/percentage of them that are equivalent/duplicated

Program	Tests	Overhead	IR					
			#M	#E	E%	#D	D%	#NEND
factor	31	20.92	599	60	10.0	64	10.7	475
head	85	157.38	2611	174	6.7	312	11.9	2125
seq	37	431.02	4441	362	8.2	595	13.4	3484
stat	68	562.50	7127	375	5.3	934	13.1	5818
unexpand	38	36.53	1166	54	4.6	214	18.4	898
Overall	259	1208.35	15944	1025	6.4	2119	13.3	12800

Table 3: Raw mutation scores and NEND mutation scores

Program	SRC		IR	
	Raw	NEND	Raw	NEND
factor	52.5	53.0	20.4	24.8
head	44.0	44.7	8.0	9.7
seq	56.4	58.5	16.2	20.1
stat	16.4	16.5	5.1	5.9
unexpand	65.0	66.4	18.3	22.6
Overall	38.8	40.1	10.2	12.2

We also compute the number/percentage of equivalent/duplicated mutants for each of the SRC and IR mutants separately. Equivalent mutants are mutants that are the same as the original program. Duplicated mutants are mutants that are equivalent to one another but not necessarily equivalent to the original program. All equivalent mutants should be ignored, while all but one mutant from an equivalence class of duplicated mutants should be ignored. The remaining mutants are then what we consider *non-equivalent, non-duplicated* (NEND) mutants [10]. We detect equivalent/duplicated mutants using trivial compiler equivalence [16]. We show these number/percentage of equivalent/duplicated mutants and the number of NEND mutants for SRC and IR in Tables 1 and 2. Even when considering only NEND mutants, there are many more mutants at the IR level than at the SRC level.

Table 3 shows the raw mutation scores (considering all covered mutants) and the mutation scores with only NEND mutants at the SRC and IR level. We see that mutation scores at SRC tend to be higher than scores at IR. These differences show the value of the tool in enabling research that asks interesting questions about mutation testing at the different levels.

Lastly, to understand the efficiency of *SRCIROR*, we measure the time overhead for mutant generation at both the SRC and IR levels. If the code is built from scratch for every mutant, the overhead of *SRCIROR* would be equal to the number of mutants generated, i.e., 4261X for SRC and 15944X for IR. However, one can use various optimizations to improve *SRCIROR*'s performance. In our evaluation, we use a simple setup that performs incremental compilation between mutants. Columns marked "Overhead" in Tables 1 and 2 show the overheads as the ratio of the time needed to generate all mutants for a given program to the time needed to perform a clean build of that program (without mutation). The overhead varies from 18.02X to 562.50X. Overall, *SRCIROR* incurs an overhead of 328.14X for 4261 SRC mutants and 1208.35X for 15944 IR mutants; overheads are much lower than the number of mutants.

4 CONCLUSIONS

We present *SRCIROR*, a toolset for performing mutation testing at the `C/C++` source and LLVM IR levels. We evaluate *SRCIROR* on a subset of programs from Coreutils and how SRC and IR compare in terms of number of mutants generated, mutation score, and number of equivalent and duplicated mutants. *SRCIROR* opens the door for performing mutation testing research for `C/C++` programs on multiple levels and comparing them.

ACKNOWLEDGMENTS

We thank Darko Marinov and the anonymous reviewers for their constructive feedback. This work is partially supported by the NSF grants CCF-1409423 and CCF-1421503.

REFERENCES

[1] Libtooling. http://clang.llvm.org/docs/LibTooling.html.
[2] Matching the Clang AST. http://clang.llvm.org/docs/LibASTMatchers.html.
[3] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *ICST*, pages 21–30, 2014.
[4] James H. Andrews and Amin Alipour. MutGen tool. https://github.com/alipourm/cmutate.
[5] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, 2005.
[6] Marcio Eduardo Delamaro and Jose Carlos Maldonado. Proteum tool for mutation testing of C programs. https://github.com/magsilva/proteum.
[7] Márcio Eduardo Delamaro and José Carlos Maldonado. Proteum—A tool for the assessment of test adequacy for C programs. In *PCS*, pages 79–95, 1996.
[8] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. Measuring effectiveness of mutant sets. In *ICSTW*, pages 132–141, 2016.
[9] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. 2018.
[10] Farah Hariri, August Shi, Hayes Converse, Sarfraz Khurshid, and Darko Marinov. Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level. In *ISSRE*, pages 105–115, 2016.
[11] Yue Jia. Milu: A higher order mutation testing tool. https://github.com/yuejia/Milu.
[12] Yue Jia and Mark Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *TAIC PART*, pages 94–98, 2008.
[13] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5):649–678, 2011.
[14] René Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *ISSTA*, pages 433–436, 2014.
[15] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava: a mutation system for Java. In *ICSE*, pages 827–830, 2006.
[16] Mike Papadakis, Yue Jia, Mark Harman, and Yves 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.
[17] Goran Petrovic and Marko Ivankovic. State of mutation testing at google. In *ICSE SEIP*, 2018.
[18] Eric Schulte. llvm-mutate. http://eschulte.github.io/llvm-mutate/.
[19] Eric Schulte. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution*. PhD thesis, University of New Mexico, 2014.
[20] Marcelo Sousa and Alper Sen. Generation of TLM testbenches using mutation testing. In *CODES+ISSS*, pages 323–332, 2012.