Mart: A Mutant Generation Tool for LLVM

Thierry Titcheu Chekam
thierry.titcheu-chekam@uni.lu
SnT, University of Luxembourg
Luxembourg

Mike Papadakis
michail.papadakis@uni.lu
SnT, University of Luxembourg
Luxembourg

Yves Le Traon
yves.letraon@uni.lu
SnT, University of Luxembourg
Luxembourg

ABSTRACT

Program mutation makes small syntactic alterations to programs' code in order to artificially create faulty programs (mutants). Mutants creation (generation) tools are often characterized by their mutation operators and the way they create and represent the mutants. This paper presents *Mart*, a mutants generation tool, for LLVM bitcode, that supports the fine-grained definition of mutation operators (as *matching rule - replacing pattern* pair; uses 816 defined pairs by default) and the restriction of the code parts to mutate. New operators are implemented in *Mart* by implementing their matching rules and replacing patterns. *Mart* also implements in-memory *Trivial Compiler Equivalence* to eliminate equivalent and duplicate mutants during mutants generation. *Mart* generates mutant code as separated mutant files, meta-mutants file, weak mutation and mutant coverage instrumented files. *Mart* is publicly available (https://github.com/thierry-tct/mart). *Mart* has been applied to generate mutants for several research experiments and generated more than 4,000,000 mutants.

CCS CONCEPTS

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

KEYWORDS

Mutation, mutant operators, software analysis, LLVM bitcode, Trivial Compiler Equivalence

ACM Reference Format:
Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2019. Mart: A Mutant Generation Tool for LLVM. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3338906.3341180

1 INTRODUCTION

Mutation testing [2] is a fault-based testing technique that is receiving more and more adoption among practitioners [8, 12]. It has been shown to be effective at finding faults in software [3]. The mutation testing process involves seeding artificial faults, called mutants, into a program under test (mutant generation), then, the mutants are executed with some test suites in order to evaluate the tests (mutants execution). The mutants generation process systematically applies a set of mutation operators (syntactic code transformation rules) on the specific code parts of interest. This process often leads to mutants that are semantically equivalent to the program being mutated (original program), called equivalent mutants, and mutants semantically equivalent with others mutants, called duplicate mutants [10].

This paper present *Mart*, a tool that generates mutants for LLVM bitcode [7] (high-level languages such as C and C++ are compiled to LLVM intermediate representation for optimization and analysis). Generating mutants at the bitcode level may lead to inconsistency with source code mutation (due to loss of structural information during compilation). Nevertheless, the major advantage of generating mutants at the LLVM bitcode level is the ability to generate mutants for multiple high-level programming languages with the same tool. Two LLVM bitcode mutation tools, namely MuLL [4] and SRCIROR [5], have been developed recently but they currently support only few mutation operators and provides limited flexibility of mutation operators configuration during mutant generation. MuLL implements Arithmetic Operator Replacement, Condition Negation, Function Call Deletion and Replacement with Constant, Scalar Value Replacement. SRCIROR implements Arithmetic Operator Replacement, Logical Connector Replacement, Relational Operator Replacement and Integer Constant Replacement operators. None of those tools mutate pointers.

Mart mutant generation tool provides:

- A rich set of mutation operators (fine-grained operators [6]), including operators that simulate high-level programming language's complex expressions (such as left increment).
- An in-memory implementation of Trivial Compiler equivalence (TCE) [10] to eliminate equivalent and duplicate mutants.
- A simple description language for mutation operators configuration. The language enables users to apply a mutation operator based on the class of the operands of the mutated code's operation.
- Generation of separated mutant bitcode files, meta-mutants bitcode file (useful for some mutant execution techniques [15]), weak mutation instrumented bitcode file and, mutant coverage instrumented bitcode file.

Mart has been used to generate mutants for research experiments [11, 14] and, it generated 4,778,157 mutant and detected 2,173,508 equivalent and duplicate mutants.

2 *MART* MUTANTS GENERATION

Mart generates mutants for LLVM bitcode programs. *Mart* takes as input an LLVM bitcode file and optionally mutation configuration files to automatically generate mutated LLVM bitcode files.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3341180

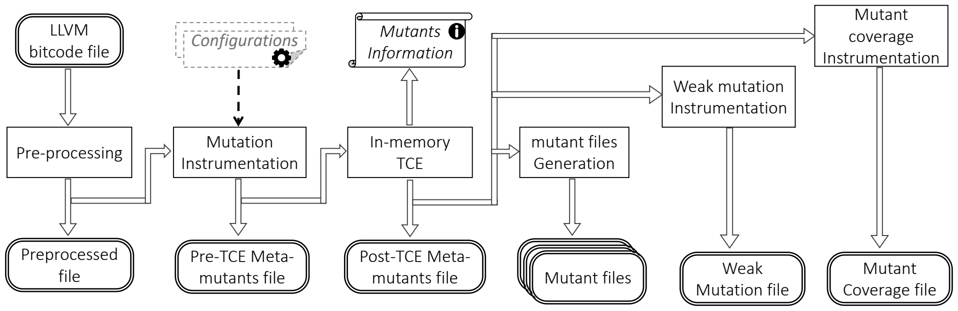


Figure 1: LLVM bitcode mutation process of *Mart*. The rounded edge rectangles with double border lines represent LLVM bitcode files. The square edge rectangles represent the steps of the mutation process. Each step is implemented by a component of *Mart*.

An overview of the process implemented by *Mart*, to generate mutants, is represented in Figure 1. Initially, the input LLVM bitcode file is pre-processed (re-formatted to ease mutant instrumentation) and then instrumented by transforming the code using mutation operators (the instrumentation can be constrained using mutation configurations). The instrumentation results in a meta-mutants program that encodes all mutants in a single module. The Meta-mutants module is then further processed by eliminating equivalent and duplicate mutants using an in-memory implementation of the Trivial Compiler Equivalence (TCE) [10]. Equivalent mutants are mutants that are semantically equivalent to the original program while duplicate mutants are mutants that are semantically equivalent to other mutants. The TCE elimination results in another meta-mutants module where equivalent and duplicate mutants, detectable by TCE, are removed. Information about mutants, such as mutant type, etc, are also exported. Finally, the post-TCE meta-mutants module is used to generate separated mutants files for different mutants, weak mutation instrumented module (to measure mutants infection) and mutant coverage module (to measure mutants reachability). The generated mutant files and instrumented files can be input to third-parties testing frameworks to be executed with test suites or improve the test suites. We recommend to use *Muteria*[1] testing framework for test execution (see the guide on *Mart* webpage).

In the following sub-sections, we present details about the implementation of components of *Mart*.

2.1 Preprocessing

The input LLVM bitcode file loaded as LLVM Module is transformed to enable the instrumentation with mutation operators. In this phase, the *phi nodes* of LLVM intermediate language are removed by applying a customized *reg2mem* function (which replaces registers by local variables). *Phi nodes* enable LLVM registers to be assigned and used in different basic blocks. This feature hinders *Mart* instrumentation as the instrumentation changes a single basic block at the time. The pre-processing step replaces registers with local variables for *phi nodes* by declaring, for each *phi node* register, a local variable which is assigned the register's value at the register writing basic block and, the variable is loaded and the value used in the register reading basic block instead of the register.

2.2 Mutation Instrumentation

The mutation instrumentation of *Mart* consists of applying the defined mutation operators on compatible code locations. In this step, a configuration of the set of mutation operators to apply as well as a configuration of the code locations to apply those mutation operators can be used to constrain the mutation.

2.2.1 Mutation Operators Representation. In order to support the mutation of complex operators of source code (e.g. recognizing C language arithmetic left increment (*++i*) or pointer de-reference followed by right decrement (**p–*) on the LLVM bitcode level), we define abstractions of mutant operators.

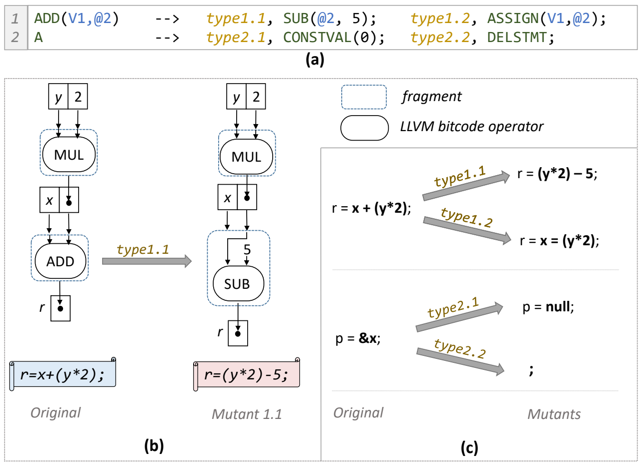
Definition 2.1. We define as code *fragment* any piece of code that can be expressed as a function. Regarding a mutation, a *fragment* is the minimal piece of LLVM code that is syntactically changed by the mutation. This code may input LLVM registers or constant values and return some value into another register.

Definition 2.2. A *fragments f'* is compatible with another fragment *f* if and only if *f* can be replaced by *f'* without breaking the code's syntax.

Mart represents each mutation operator as a pair of *fragments* (f, f'), where f' is compatible with f. To apply the operator (f, f') at a location l of a program P, the *fragment* f is matched then, if found, it is replaced by the *fragment* f'. the resulting program M after replacing f by f' at l on P is a mutant of P. Figure 2-(b) illustrate an example of a mutation as executed by *Mart*.

Implementing New Mutation Operators. Given that *fragments* need to be matched and/or replaced, *Mart* provides an interface to implement *fragments* where matching and replacing functions need to be implemented. Implementing a new operator requires to implement the *fragments'* interfaces. The matching function inputs a list of LLVM bitcode instructions, checks whether the *fragment* is matched or not and, returns the *fragment* input addresses and output register address. The replacing function input the list of matched *fragment*'s inputs and the list of bitcode instructions to mutate then, replace the code instructions to mutate.

[1] https://github.com/muteria/muteria



```
ADD(V1,@2)    -->   type1.1, SUB(@2, 5);    type1.2, ASSIGN(V1,@2);
A             -->   type2.1, CONSTVAL(0);   type2.2, DELSTMT;
```

Figure 2: Example of bitcode mutation by *Mart*. Sub-figure (a) is an example of a mutation operators configuration description in a simple description language (see section 2.2.3). Sub-figure (b) illustrates an example of code mutation; the second *fragment* in the original code is replaced by a mutant *fragment* (see section 2.2.1). Sub-figure (c) presents an example of the mutation using the configuration of (a).

2.2.2 Currently Supported Mutation Operators. Currently, *Mart* implements 18 operator groups (pairs of "compatible" fragment groups). The 18 operator groups are designed to match a large number of elements of program syntax (additional operator groups can be implemented). There are 68 *fragments* implemented and the default mutation configuration is made of 816 operators (pair of *fragments*), including variations due to operand classes (see section 2.2.3). These include all those that are supported by modern mutation testing tools [9]. The 18 operator groups are recorded in Table 1. "Original *fragment* group" refers to the matched *fragment* and "mutant *fragment* group" refers to the replacing *fragment*.

The *fragment* groups are defined as following (p refers to pointer values and s refers to scalar values):

- **ANY STMT** refers to matching any type of statement (only original *fragment*).
- **TRAPSTMT** refers to a *trap*, which cause the program to abort its execution (only mutant *fragment*).
- **DELSTMT** refers to the empty statement, thus, replacing by this is equivalent to deleting the original statement (only mutant *fragment*).
- **CALL STATEMENT** refers to a function call.
- **SWITCH STATEMENT** refers to a C language like *switch* statement.
- **SHUFFLEARGS** refers to the same function call as the original, with arguments of same type swapped (e.g. $g(a, b) \rightarrow g(b, a)$). This can only be a mutant *fragment* and, requires the original *fragment* to be a function call.
- **SHUFFLECASESDEST** refers to the same *switch* statement as the original, with the basic blocks of the *cases* swapped (e.g. $\{case\ a : B_1;\ case\ b : B_2;\ default : B_3; \} \rightarrow \{case\ a : B_2;\ case\ b : B_1;\ default : B_3; \}$). This can only be used as mutant *fragment* and, requires the orignal *fragment* to be a *switch* statement.
- **REMOVECASES** refers to the same *switch* statement as the original, with some *cases* deleted (the corresponding values will lead to execute the *default* basic block) (e.g. $\{case\ a : B_1;\ case\ b : B_2;\ default : B_3; \} \rightarrow \{case\ a : B_2;\ default : B_3; \}$). This can only be used as mutant *fragment* and, requires the orignal *fragment* to be a *switch* statement.
- **SCALAR.ATOM** refers to any non pointer type variable or constant (only original *fragment*).
- **POINTER.ATOM** refers to any pointer type variable or constant (only original *fragment*).
- **SCALAR.UNARY** refers to any non pointer unary arithmetic or logical operation (e.g. $abs(s)$, $-s$, $!s$, $s++$ ...).
- **POINTER.UNARY** refers to any pointer unary arithmetic operation (e.g. $p++$, $--p$ ...).
- **SCALAR.BINARY** refers to any non pointer binary arithmetic, relational or logical operation (e.g. $s_1 + s_2$, $s_1 \&\& s_2$, $s_1 >> s_2$, $s_1 <= s_2$ ...).
- **POINTER.BINARY** refers to any pointer binary arithmetic or relational operation (e.g. $p + s$, $p_1 > p_2$ ...).
- **DEREFERENCE.UNARY** refers to any combination of pointer dereference and scalar unary arithmetic operation, or combination of pointer unary operation and pointer dereference (e.g. $(*p)--$, $*(p--)$ ...).
- **DEREFERENCE.BINARY** refers to any combination of pointer dereference and scalar binary arithmetic operation, or combination of pointer binary operation and pointer dereference (e.g. $(*p) + s$, $*(p + s)$ ...).

Table 1: Mutant Types

Mutated Code	Original *Fragment* Group	Mutant *Fragment* Group
STATEMENT	ANY STMT	TRAPSTMT
	ANY STMT	DELSTMT
	CALL STATEMENT	SHUFFLEARGS
	SWITCH STATEMENT	SHUFFLECASESDESTS
	SWITCH STATEMENT	REMOVECASES
EXPRESSION	SCALAR.ATOM	SCALAR.UNARY
	SCALAR.ATOM	SCALAR.BINARY
	SCALAR.UNARY	SCALAR.UNARY
	SCALAR.BINARY	SCALAR.UNARY
	SCALAR.BINARY	SCALAR.BINARY
	SCALAR.BINARY	TRAPSTMT
	SCALAR.BINARY	DELSTMT
	POINTER.ATOM	POINTER.UNARY
	POINTER.UNARY	POINTER.UNARY
	POINTER.BINARY	POINTER.UNARY
	POINTER.BINARY	POINTER.BINARY
	DEREFERENCE.BINARY	DEREFERENCE.UNARY
	DEREFERENCE.BINARY	DEREFERENCE.BINARY

2.2.3 Instrumentation process. *Mart* mutation instrumentation visits the Control Flow Graph (CFG) of the module under mutation and for each statement (represented by a group of instructions that are data-dependent w.r.t. registers) l, create mutated versions $l'_1, .., l'_k$.

A branching instruction is then inserted, to select, based on the value of a special global variable called "Mutant ID selector", the statement to execute between $l, l'_1, .., l'_k$. The resulting module is a meta-mutants module where, the module can represent a specific mutant by just setting the value of the "Mutant ID selector" variable to its ID.

Constrained Mutation. The instrumentation process is subject to possible configuration. Users can restrict the corresponding source code's source files and functions to mutate by specifying the values in a JSON file that is used during mutation instrumentation. The operators to apply can also be specified in a file where each line is a key-value with the key the matching pattern and value the list or replacing patterns (This makes a simple mutation operator description language). Each pattern is made of the *fragment* name and the list of its indexed arguments' classes. The arguments classes are *constant (C)*, *scalar variable (V)*, *address (A)*, *pointer variable (P)* and *any expression (@)*. The set of argument classes in the replacing pattern is a subset of those from the matching pattern (except for constants). Refer to the tool web page[2] for more details on the mutation operator description language. The mutation operation configuration file can easily be created automatically with a script available with the tool. Figure 2 shows an example of a mutant operator description configuration where 4 mutation operators are defined (2 operators for matching the sum of a variable and any expression and, 2 for matching an address).

2.3 In-Memory Equivalent and Duplicate Mutants Elimination

Mart eliminates equivalent and duplicate mutants by applying Trivial Compiler Equivalence (TCE) [10], on the mutated functions, in-memory. Our implementation of TCE applies LLVM optimizations, for each mutant, to the mutated function and uses a customized version of llvm-diff[3] tool to check for the difference between the optimized functions of the mutants and the original program's.

2.4 Final Mutated Files Generation

After TCE equivalent and duplicate mutants have been eliminated from the meta-mutants module, separated mutants files are generated by dumping the mutants functions used during TCE (which are also linked with the un-mutated function to make complete mutant bitcode). weak mutation and mutants coverage instrumented bitcode modules are generated by replacing, in the meta-mutants module, each mutant's code by label (function call that writes, into a file, the mutant ID of the mutants whose label is covered during test execution). A mutant coverage label is covered by any test that reaches it (the mutant location). A weak mutation label is covered by any tests that infect the mutant.

3 IMPLEMENTATION AND USAGE

Mart is implemented as a static analysis tool for LLVM bitcode (*Mart* loads the input LLVM bitcode file as an LLVM module and manipulates the module using the LLVM API). Currently, *Mart* has been tested for LLVM versions 3.4, 3.7, 3.8 and 3.9, and on Ubuntu (Linux) operating system.

[2]https://github.com/thierry-tct/mart
[3]https://llvm.org/docs/CommandGuide/llvm-diff.html

Table 2: *Mart* in Practice

Benchmark	# of Programs	# Generated Muts.	# TCE Eq./Dup. Muts.
CoREBench [1]	46	1,564,614	715,996
Codeflaws [13]	1,692	3,213,543	1,457,512

Mart can be used to mutate programs written in any language compilable into LLVM bitcode (compile complex C/C++ projects with wllvm[4]). The users are required to compile the code with debug information enabled in order to keep the information about source code location for mutants information. If no debug information is found in the program to mutate, the mutants information will not contain the source code locations information of the mutants.

Mart can be used through the command lines interface (CLI) or through its application programming interface API. Users of *Mart* can provide mutation configuration files (mutants operators and mutation scope), decide whether to apply in-memory TCE, decide whether to output weak mutation bitcode file, mutant coverage bitcode file and separated mutant files. A demonstration video is available online[5]. See the tool weblink to get started.

4 EXPERIMENTING WITH *MART*

Mart has already been used to generate mutants for 2 research experiments [11, 14]. The experiments applied *Mart* on a set of C language projects. The experiments (shown in table 2) generated a total of more than 4 million mutants and, in-memory TCE detected more than 2 million equivalents and duplicate mutants.

5 CONCLUSION

Mart mutants generation tool is designed to provide flexibility to configure mutants generation and the possibility to design bitcode mutation that can capture some of the semantics of source code mutation of complex operations. *Mart* provides a simple format for the description of mutants operators to apply on bitcode, a large number of implemented mutation operators, an interface to implement new operators, and an in-memory trivial compiler equivalence to detect equivalent and duplicate mutants. *Mart* is designed to only generate and compile mutants from LLVM bitcode, leaving the mutant-test execution to the user. This choice gives flexibility to the user on the test execution framework to use. *Mart* has been used to conduct research experiments and successfully generated a large number of mutants on real software. Thus, *Mart* can be used by both researchers and practitioners. *Mart* has the same limitation of all bitcode mutation tools, which is the inability to precisely mutate specific features of higher level languages.

Future plans on the development of *Mart* involve implementing additional operator groups to handle more program syntax elements such as loops. *Mart* is made publicly available, open-source with MIT License, at the following web link https://github.com/thierry-tct/mart

ACKNOWLEDGMENTS

Thierry Titcheu Chekam is supported by the AFR PhD Grant of the National Research Fund, Luxembourg.

[4]https://github.com/travitch/whole-program-llvm
[5]https://youtu.be/V2Hvi_iqiVE

REFERENCES

[1] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: studying complexity of regression errors. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 105–115. https://doi.org/10.1145/2610384.2628058

[2] Timothy Alan Budd. 1980. *Mutation Analysis of Program Test Data*. phdthesis. Yale University, New Haven, Connecticut.

[3] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *ICSE 2017*. 597–608.

[4] A. Denisov and S. Pankevich. 2018. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 25–31. https://doi.org/10.1109/ICSTW.2018.00024

[5] Farah Hariri and August Shi. 2018. SRCIROR: A Toolset for Mutation Testing of C Source Code and LLVM Intermediate Representation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 860–863. https://doi.org/10.1145/3238147.3240482

[6] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring mutant utility from program context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 284–294. https://doi.org/10.1145/3092703.3092732

[7] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[8] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. 2017. Assessing and Improving the Mutation Testing Practice of PIT. In *ICST 2017*. 430–435.

[9] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Trans. on Soft. Eng. & Meth.* 5, 2 (April 1996), 99–118.

[10] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *ICSE 2015*. 936–946.

[11] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. 2018. Mutant Quality Indicators. In *the 13th International Workshop on Mutation Analysis (Mutation 2018)*.

[12] Goran Petrovic and Marko Ivankovic. 2018. State of Mutation Testing at Google. In *40th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2018, May 27 - 3 June 2018, Gothenburg, Sweden*.

[13] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *ICSE 2017*. 180–182.

[14] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé Bissyandé, Yves Le Traon, and Koushik Sen. 2018. Selecting Fault Revealing Mutants. *arXiv preprint arXiv:1803.07901v3* (2018).

[15] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster mutation analysis via equivalence modulo states. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 295–306. https://doi.org/10.1145/3092703.3092714