

Mull it over: mutation testing based on LLVM

Alex Denisov
Independent researcher
Berlin, Germany
Email: alex@lowlevelbits.org

Stanislav Pankevich
Independent researcher
Berlin, Germany
Email: s.pankevich@gmail.com

Abstract—This paper describes Mull, an open-source tool for mutation testing based on the LLVM framework. Mull works with LLVM IR, a low-level intermediate representation, to perform mutations, and uses LLVM JIT for just-in-time compilation. This design choice enables the following two capabilities of Mull: language independence and fine-grained control over compilation and execution of a tested program and its mutations. Mull can work with code written in any programming language that supports compilation to LLVM IR, such as C, C++, Rust, or Swift. Direct manipulation of LLVM IR allows Mull to do less work to generate mutations: only modified fragments of IR code are recompiled, and this results in faster processing of mutated programs. To our knowledge, no existing mutation testing tool provides these capabilities for compiled programming languages. We describe the algorithm and implementation details of Mull, highlight current limitations of Mull, and present the results of our evaluation of Mull on real-world projects such as RODOS, OpenSSL, LLVM.

Index Terms—mutation testing, llvm

I. INTRODUCTION

Mutation Testing, a fault-based software testing technique, serves as a way to evaluate and improve quality of software tests. A tool for mutation testing creates many slightly modified versions of original program and then runs a test suite against each version, which is called a *mutant*. A mutant is said to be *killed* if the test suite detects a change to the program introduced by this mutant, or *survived* otherwise. Each mutation of original program is created based on one of the predefined rules for program modification called *mutation operators*. Each mutant is represented by a *mutation point*: a combination of mutation operator and location of a mutation in the program’s source code. To assess the quality of a test suite mutation testing uses a metric called *mutation score*, or *mutation coverage*.

Mutation testing is getting interest from the open source community. More and more open-source mutation testing tools targeting various programming languages appear [1]. Unfortunately, not all of these tools reach a level of maturity needed for practical use. While mature implementations of open-source mutation testing tools definitely exist, with Pitest [2] and Mutant [3] being strong examples from Java and Ruby programming language communities, there is still a lack of usable mutation testing tools for certain compiled programming languages.

In this paper, we present the Mull project, our attempt to build a general-purpose mutation testing tool targeting compiled languages. Mull is built on top of the LLVM compiler

framework [4]. It uses two components of LLVM: IR, its low-level intermediate language, to perform mutations and JIT for runtime compilation and execution of a tested program and its mutated counterparts. LLVM IR is also referred to as *LLVM Bitcode* or simply as *bitcode*. We use these terms interchangeably.

We consider the following criteria important for a practical implementation of mutation testing tool: the tool must be fast, configurable and easy to set up and use. The tool should allow smooth integration with build tools. The tool should be ready for use in mutation testing analysis of real-world production and open source projects. The tool should implement a reasonable number of basic mutation operators to enable the practical use of it in different domains such as systems programming, application programming, algorithms and mathematical computations.

Mull is built with all of the above criteria in mind. We started Mull with a primary focus on C and C++, but due to LLVM, Mull can work with any other programming language that compiles to LLVM IR, such as Rust, Swift, Objective-C. To add a language support one needs to implement adapters to the test frameworks used by the programming language.

Mull is a command line tool. It takes a configuration file as an input and produces an SQLite database with the results as output. Configuration options include a list of tested program’s bitcode files, a set of mutation operators, a test framework, and a few other settings. The SQLite database contains information that Mull gathers while running on a tested program, such as tests, mutation points, mutants (killed or survived), and more. As a command-line tool, Mull does not show mutation score or mutation coverage. There is a separate program that generates an HTML report from the SQLite file.

Mull’s source code is available online [5] under Apache License, version 2.0 [6].

We organize the rest of the paper as follows. Section II describes the algorithm of Mull. Section III then goes deeper and describes what we consider the most interesting implementation details of Mull. Section IV describes the mutation operators currently implemented in Mull. Section V describes our evaluation of the open source projects: RODOS, OpenSSL, LLVM. Section VI highlights the limitations of Mull. Section VII discusses future work. Section VIII concludes the paper.

II. ALGORITHM

The following are the steps that Mull performs during a session:

Step 1: Mull loads LLVM Bitcode into memory.

Step 2: Mull inserts instrumentation code into each function. This code is used to collect code coverage information. We describe our approach to instrumentation in III.A.

Step 3: Mull compiles instrumented LLVM Bitcode to machine code and prepares the machine code for execution by LLVM JIT engine.

Step 4: In the LLVM IR code Mull finds the tests according to a test framework specified in the configuration file.

Step 5: Mull runs each test using LLVM JIT engine and collects code coverage information.

Step 6: Mull finds mutations in the LLVM IR code based on a code coverage information collected for each test. A set of mutation points is created.

Step 7: For each mutation point, Mull creates a mutant and runs each test that can kill the mutant. For each mutant, only part of bitcode is recompiled into machine code. We describe our approach to runtime compilation in III.B.

Step 8: All information collected during the session is written to the SQLite database. This is the final step. Mull finishes its execution at this point.

III. IMPLEMENTATION

A. Instrumentation and Dynamic Call Tree

A typical program has many mutations, but not all of them are reachable by the program's tests. We use this fact to reduce the number of mutants. To know which mutations are reachable we thus need to know which code is reachable from a test. To achieve this, we insert instrumentation into each function and then run a test to gather code coverage information. From this information, we construct a *dynamic call tree*.

The purpose of the call tree is better illustrated by example. Consider a function `test_driver` calling function `test` which is calling functions `testee1` and `testee2`. The call tree would look like the following: `test_driver -> test -> { testee1, testee2 }`. In this case the code being tested is inside of `testee1` and `testee2`. Therefore we can inject mutations only into the subtrees of the `test` function.

The call tree adds more fine-grained control of the amount of work via *mutation distance*. We can define a mutation distance to be a distance from a test function to a function with the actual mutation. If function A calls function B and function B calls function C, then the distance between A and C is 2. Mutation distance can be used to decrease the number of mutations: Mull can ignore mutations that are too far from a test function.

The instrumentation-based approach has an overhead, but it is necessary to get the right code coverage information. Initially, we used static code analysis to build the call tree: Mull iterated through bitcode and followed call instructions to

build the tree. Unfortunately, a function is not always known until runtime. Typical examples are C function pointers and C++ virtual function calls. After a few failed attempts we switched to the dynamic instrumentation. Thus the *call tree* became *dynamic call tree*.

B. JIT and Runtime Compilation

To run a tested program, Mull needs to compile the bitcode files into object files containing machine code and link them together into executable, as any compiler would do. To accomplish this task Mull utilizes LLVM JIT engine. This approach has a great advantage: compilation and linking happen in memory. Thus there is no disk I/O overhead.

When it comes to mutation Mull performs it on a copy of a single bitcode file, recompiles it and links together with already compiled object files. Partial recompilation helps to increase performance. It also helps to decrease memory usage: mutated bitcode file can be disposed from memory right after execution.

C. Sandboxing

Mutations can make the code behave in unexpected ways: to crash, to timeout or to exit prematurely. We use a parent/child process isolation to achieve a proper sandboxing of a tested program.

Mull, which is a parent process, runs each test in a separate child process. The `fork` system call is used to create a child process, `mmap` system call is used to share memory between the parent process and the child process.

Mull handles exit status of a child process according to the following policy:

1) *Normal execution (test has passed or failed):* We use a conventional exit code 227 to indicate if a test has run without any issues. If child process exits with code 227, Mull knows that a test has either succeeded or failed and that nothing extraordinary like in one of the following cases has happened.

2) *Timeout:* Mutated code might never finish its execution in a child process. To handle this case Mull sets an alarm in a child process that exits with a conventional timeout code 239 after a certain time interval. We use `ualarm` function to set the alarm.

3) *Crash:* Mutated code can crash with a child process executing it. We use `WIFSIGNALED()` to detect a crash of a child process.

4) *Abnormal exit:* Mutated code exits prematurely from a child process and this does not let a test to finish. This scenario is a reason for the existence of the custom exit code 227 from the case 1) because Mull needs to distinguish between normal exit and abnormal exit from a child process.

D. Dry Run

It is not known in advance how many mutations a project has and how much time does it take Mull to run it. To remove uncertainty, we introduce *dry run* mode. In this mode, Mull collects information about mutations but does not run tests against them. Therefore no partial recompilation and no sandboxing are needed.

Additionally, Mull gives a pessimistic approximation of the run time: it calculates how much time would be needed if each mutant times out. Real execution time is lower than the approximation, but it gives a good hint of expected run time.

E. Test Framework: plugin architecture

Mull can work with any test framework. The only requirement is that Mull can run a single test independently from the other tests in a test suite.

Each test framework plugin consists of two components: *test finder* and *test runner*. Mull uses test finder to find the tests in a bitcode of a tested program, and it uses test runner to run one test according to the calling conventions of a given test framework.

Test finder takes all bitcode files as an input and gives back a list of test functions. Examples: for `SimpleTestFinder` a test is simply a C function whose name starts with `test`, for `GoogleTestFinder` a search algorithm extracts the information about the test functions from `internal::MakeAndRegisterTestInfo` registration call of a `GoogleTest` framework.

Test runner runs one test and returns the result of its execution. Running a test can be as easy as calling a test function and checking its return value: for `SimpleTestRunner` test passes if its test function returns 1 and fails if it returns 0. For `GoogleTest` framework `GoogleTestRunner` has to emulate the work of `GoogleTest`'s `main()` function: to run one test `GoogleTestRunner` runs a test suite in a "focused mode" with a filter set to a name of this test's function (`--gtest_filter=TestFunctionName`).

Many projects have their custom test suites. Examples are `Musl`, `OpenSSL`, `glibc`, `openlibm`. While it is possible to create a dedicated pair of test finder and test runner for each of these projects like `OpenSSLTestFinder` and `OpenSSLTestRunner`, we created a general solution called `CustomTestFramework` to enable testing of these projects. To use `CustomTestFramework` with a given project one has to provide the custom test definitions in a configuration file. Here is an example for `OpenSSL` project:

```
test_framework: CustomTest
custom_tests:
- name: test_bio_enc_aes_128_cbc
  method: test_bio_enc_aes_128_cbc
  program: bio_enc_test
  arguments: [ test_bio_enc_aes_128_cbc ]
```

In this case `CustomTestFinder` treats a function called `test_bio_enc_aes_128_cbc` as a test, and `CustomTestRunner` runs the program using specified arguments.

F. Fail Fast mode

In the worst case a tested program with N tests and M mutations requires $N * M$ test runs. Mull has an option to decrease the amount of test runs: *fail fast* mode. For example, if a mutation is reached from 20 tests and the very first test

kills the mutant, then there is no need to run the remaining 19 tests. The fail fast mode is disabled by default and can be enabled in the configuration file.

G. Caching

Mull uses JIT and Runtime Compilation for better performance. However, sometimes it is faster to read object file from disk than to compile it in memory from LLVM Bitcode. In this regard, Mull supports on-disk cache. Before compiling a bitcode file Mull attempts to get an object file from disk. If there is none, then Mull compiles the bitcode file and saves resulting object file on-disk for later usage. When Mull runs next time, it can use an object file from the previous session.

To avoid a use of outdated object files, Mull encodes checksum of original bitcode file into the name of a cached object file. Object files for mutants also contain a unique identifier of the mutation point.

IV. SUPPORTED MUTATION OPERATORS

Mull performs mutations on the LLVM IR code, so its implementation of mutation operators is largely determined by the specification of LLVM language and in particular its Instruction Reference [7]. We also used Pitest's documentation of its mutation operators [8] to decide which operators to implement first.

The following is the list of mutation operators currently supported by Mull. All of the instructions referenced below can be found in the LLVM IR language manual.

A. Math: Add, Sub, Mul, Div

This group of operators performs mutations of basic arithmetic operators: "+" to "-", "-" to "+", "*" to "/", "/" to "*".

Math Add replaces an add instruction, which returns the sum of its two operands, with a sub instruction, which returns the difference of its two operands. Math Add also works with the floating equivalent of add instruction: `fadd` which is replaced with `fsub`.

Math Sub operator performs the same kind of mutation as Math Add but in opposite direction: from sub to add and `fsub` to `fadd`. Math Mul and Div work with `mul`, `fmul` and `div`, `fdiv` instructions respectively.

B. Negate Condition

Negate Condition operator works with `icmp` instruction (comparison of integer operands) and `fcmp` instruction (comparison of floating-point operands). Both instructions accept three operands of which "the first operand is the condition code indicating the kind of comparison to perform". This first operand is a conventional code that represents a type of comparison, for example: "unsigned equal" to "signed less than". Negate Condition modifies the code to achieve a complete negation of a condition: from "equal" to "not equal", from "signed less" to "signed greater than or equal", etc.

C. Remove Void Function

This operator removes a call to a void function from LLVM IR code. The void function calls can be represented by two instructions in LLVM IR: `call` and `invoke`. The difference between these instructions is related to the details of exception handling and is hidden well behind LLVM IR API making this difference irrelevant to Mull.

D. Replace Call

This operator replaces a function call, whose return value is an integer or floating-point scalar value, with an arbitrary value according to the following simple rule: the function call is replaced with a value forty-two (42) of a corresponding integer or floating-point type. Like Remove Void Function operator, this operator works with `call` and `invoke` instructions.

E. Scalar Value Replacement

This operator replaces an integer or floating-point scalar value with a predetermined value according to the following simple rule: non-zero value is replaced with a zero value (0) of the corresponding integer or floating-point type, zero value is replaced with a value of one (1) of the corresponding integer or floating-point type. Scalar values can appear as operands of many different instructions in LLVM IR language: binary arithmetic instructions like `add` or `mul`, comparison instructions `icmp` and `fcmp`, return instruction `ret`, function call instructions `call` and `invoke` and some others. Scalar Value operator maintains a list of such instructions that determines if a particular instruction can be a target of a Scalar Value mutation.

V. EVALUATION

In this section, we describe our experience of applying Mull on real-world projects. We focus on ease of integration, performance, and a practical applicability of Mull, rather than on concrete results such as found bugs or shallow tests. For this paper we picked three open-source projects: RODOS [9], OpenSSL [10] and LLVM [11]. Table I describes some properties of these projects. The number of lines of code represents size and scale of a project. However, more representative metric is a number and an overall weight of bitcode files: it has a direct impact on performance because all this code has to be loaded into memory, analyzed, compiled, and linked together.

Measurements for OpenSSL and LLVM were made on macOS 10.13 with 16GB of RAM and Intel i7 3.1GHz CPU. Measurements for RODOS were made on the same machine, but inside of VirtualBox running Ubuntu 16.04, 32 bit. 4GB of RAM and two cores of the Intel i7, 3.1GHz were allocated for the virtual machine.

For this experiment we used three mutation operators: Math Add (IV-A), Negate Condition (IV-B), and Remove Void Function (IV-C). All tests were run with the Fail Fast mode (III-F) and Caching (III-G) enabled. We ran each group of tests twice: a cold run, without cache in place, and a hot run, with cache in place.

TABLE I
PROJECTS

Project	Lines of code	Bitcode files	Bitcode size	Average time per test run
RODOS	125,127	32	407 KB	23 ms
OpenSSL	311,293	630	11 MB	42 ms
LLVM	1,324,567	224	242 MB	31 ms

For each project we measure how many tests a test suite has, how many mutants Mull detects given the mutation operators mentioned above, total amount of test runs executed during analysis, and the total execution time for both cold and hot runs.

A. RODOS

RODOS [9] is a real-time operating system developed by the German Aerospace Center. It is written in C and C++ and uses CppUnit test framework [12] for its test suite. Among several bare-metal platforms, it can be run on Linux and other POSIX-compliant operating systems.

RODOS has many small test suites, each of them covering very specific part of the system. Examples are: `matrix4d_test`, `quaternion_test`, `filesystem_test`, `hal_gpio_test`. Each test suite is designed to run only one single test per compilation: to run a test one has to compile the test suite enabling a test by providing a macro definition. We have to change this to control the test selection at runtime rather than at compile time. Once the test suite is compiled, it can run either all tests, or the one specified via command-line arguments (`argv`). Original test driver always exits with exit code 0. To check if tests failed or not one has to either observe the output or check a test report that is written into XML file. We have to add a small change here as well: exit code should represent amount of failed tests. If all tests pass, then exit code is 0, otherwise some positive number. This is a widely used approach. RODOS uses Makefiles to build and run its test suites, we have to change compiler flags (`CXXFLAGS`) to emit LLVM Bitcode.

Two more workarounds are required to run Mull against RODOS. Some parts of the system are written in assembly so they are compiled directly into machine code. We have to point Mull to them using `object_file_list` configuration option. Since RODOS uses CppUnit we also have to point it to the `libcppunit.so` via `dynamic_library_list` configuration option.

Once these preparations are done, we can run Mull against RODOS. For this experiment, we pick five test suites based on amount of tests in each of them. Table II contains the results.

B. OpenSSL

OpenSSL [10] is a well-known implementation of TLS and SSL protocols. It is written in C. It uses custom test framework for its tests. OpenSSL has a mix of unit and integration tests. We have to look at each test suite to identify if it is a unit test suite or an integration test suite. Test suites with unit tests are

TABLE II
RESULTS FOR RODOS

Test suite	Tests	Mutants	Test runs	Distance	Total time (cold / hot)
linkinterfaceuart	11	19	186	2	4s / 2s
stdlib_pico	10	36	356	2	5s / 4s
thread	10	57	196	3	5s / 3s
sortedlist	9	16	85	4	2s / 2s
linkinterfacecan	8	18	471	5	19s / 12s

TABLE III
RESULTS FOR OPENSLL

Test suite	Tests	Mutants	Test runs	Distance	Total time (cold / hot)
packettest	22	67	173	3	30s / 14s
dectest	20	274	1256	3	47s / 29s
test_test	19	102	214	4	31s / 17s
igettest	10	118	709	2	29s / 18s
bio_enc_test	6	708	2667	12	3m8s / 2m45s

simple programs that are compiled into an executable. Each test suite can only run all tests at once. We have to change them to run one test based on command line arguments, or to run all of them if no arguments are given, to preserve original behavior. We also have to extract information about each test manually. To set up `CustomTestFramework` Mull needs to know which function is a test and which command line arguments to pass to run this very specific test.

Obtaining LLVM Bitcode is trivial. To compile OpenSSL one has to invoke `configure` script to prepare build system. `configure` accepts additional parameters that are used as `CFLAGS`. We invoke the script by passing `-flto` to enable LTO [13], which produces bitcode files instead of object files as build artifacts. Then we compile test suites of interest and construct separate configuration files for each of them.

Results of this experiment can be found in Table III.

C. LLVM

The LLVM [11] compiler infrastructure project is the biggest project we have analyzed so far. LLVM is written in C++. It uses GoogleTest [14] as a test framework. It has several unit test suites of various sizes targeting different subsystems of LLVM. For this experiment, we use `ADTTests`: a test suite that covers specific abstract data types used in LLVM such as arrays, strings, maps, integers, floats, etc. Additionally, in this test suite, we focus only on normal tests and exclude tests which are based on Typed Tests feature of GoogleTest that `GoogleTestFinder` does not support yet.

Obtaining Bitcode is trivial for LLVM: it has a build setting that enables LTO [13], which produces bitcode files instead of object files as build artifacts. We use only one workaround to get LLVM's tests running: LLVM JIT does not support Thread-Local Storage [15] so we have to exclude one source file that uses TLS from the compilation. Fortunately, this file is not used in the test suite so its absence does not affect the analysis.

TABLE IV
RESULTS FOR LLVM

Test suite	Tests	Mutants	Test runs	Dist.	Total time (cold / hot)
All Tests	550	11779	60325	25	3h46m / 1h54m
All Tests	550	5508	13601	2	1h52m / 47m46s
APFloat	70	1894	22010	25	41m1s / 18m21s
APFloat	70	361	1622	2	14m13s / 4m32s
StringExtras	5	160	165	7	4m44s / 3m2s
StringExtras	5	93	98	2	4m36s / 2m24s

LLVM is a big project. In this case it is recommended to launch Mull in Dry Run mode (III-D) to get information about tested program. Dry run shows that Mull detected 550 tests and found 11779 mutants, it also shows that there are 60325 test runs according to III-F. The report also shows approximation of execution time: full run may take about 9 hours at maximum. It helps to see the order: hours, not days in the worst case. The approximation is very pessimistic: Mull assumes that every mutant fails because of a timeout. In fact, real execution time was 3 hours 46 minutes.

Table IV shows the results for different configurations. We use three groups of tests of different size and different mutation distance (III-A) for each of them to show applicability of Mull even for big projects. The execution time of almost four hours on the whole test suite is impractical for iterative development process as opposed to two minutes for a subset of tests.

VI. CURRENT LIMITATIONS

A. Junk and stray mutations

A mutation can exist in bitcode, but cannot be achieved by changing original source code. Such mutation is called *junk mutation*. The term was first coined by Henry Coles [16]. A good example of such mutation in C++ is a `std::vector::push_back` method call: one line of C++ code produces around 200 LLVM IR instructions. Depending on mutation operator Mull finds mutations in those instructions even though there is no equivalent in the original source code. Some mutation operators require advanced pattern matching to avoid this issue, for others, we did not find a robust solution yet.

Another issue is C and C++ code from their standard libraries. Compiler inlines code from macros and templates into resulting bitcode. Mull finds mutations in this code as well, but they are not relevant to a tested program. We call such mutations *stray mutations*. Fortunately, there is a simple workaround: Mull can filter out mutations based on their location in a source code using `exclude_locations` configuration option. This approach also helps to avoid mutation of third-party code.

B. Current limitations of LLVM JIT

Mull uses LLVM JIT from which it gets its power as well as some of its limitations. The following are two major limitations we encountered: LLVM JIT does not work with projects using Thread Local Storage [15], and it does not

support Objective-C Runtime [17]. The latter limitation is the only reason why Mull does not yet fully support Objective-C and Swift programming languages. Both problems are solvable and are waiting for their solution.

VII. FUTURE WORK

There is a lot of work to be done to get Mull closer to its use in production. Below, we outline the three major (and most obvious) parts of our work: performance improvements, integration with modern IDE's, further exploration of the real-world projects.

One direction of work is further performance optimizations: parallelization and even better control over recompilation of bitcode. Mull still runs only one child process at a time so the work with multiple child processes is one of the nearest optimizations we are planning. Recompilation of mutated function instead of a whole bitcode file that contains it can improve performance of Mull on projects with large bitcode files.

Integration with existing IDE's is yet another important part of work to make Mull practical for daily use. While Mull works perfectly as a command-line tool that produces HTML reports, we also see it natural to be a part of a workflow provided by the modern IDE's.

Another direction of work is a further exploration of the real-world projects that will drive the implementation of new test framework adapters like Catch for C++, better support of programming languages like Rust and Swift, running Mull on BSD and Windows systems. In this regard, we especially look forward to the proper support of Objective-C Runtime by LLVM JIT because it will open Mull the door to the world of desktop and mobile application development on macOS and iOS platforms.

We are aware that other mutation testing tools for compiled programming languages exist [18] and we assume that a proper comparison between Mull and these tools should be a topic of separate research.

VIII. CONCLUSION

Our choice of LLVM as a base for an implementation of a mutation testing tool was based on an experiment with LLVM IR and LLVM JIT libraries that had produced results superior to those from any of our previous attempts to implement a solution working on source code or AST levels. So far, we did not encounter a single critical problem that would turn us away from our decision to base Mull on LLVM with its intermediate language and infrastructure. Quite to the opposite, Mull satisfies all criteria that we consider important for an implementation of mutation testing tool. It has a great number of applications and a large room for further improvement.

To test Mull on real-world projects and to explore possible limitations of our approach we applied it to as many different projects, programming languages, test frameworks and operating systems as was possible with our capacity. The following is the list of the projects we analyzed:

- *LLVM* (C/C++, GoogleTest, macOS)

- *OpenSSL* (C/C++, custom test suite, macOS)
- *RODOS* (C/C++, CppUnit, Linux)
- *openlibm* (C, custom test suite, macOS)
- *newlib's libm* (C, custom test suite, Linux)
- *fnt* (C++, GoogleTest, macOS)
- *CryptoSwift* (Swift, XCTest, Linux)
- *rustc-demangle* (Rust, Rust's test framework, macOS)
- *Mull (autoanalysis)* (C++, GoogleTest, macOS)

Our long-term goal is to get Mull to the point where it can be used by industry as a drop-in solution for mutation testing. Also, we expect Mull to find a use in research, including interaction with other tools and approaches, that would find solutions to speed up the normally slow process of mutation testing with automatic test generation.

ACKNOWLEDGEMENTS

We thank Henry Coles and Markus Schirp for fruitful discussions and their helpful advice at the early stage of development of Mull.

We thank Yue Jia and Mark Harman for their Analysis and Survey [19] that gave us a theoretical background for our work.

We thank Tobias Grosser for the hint about LTO option that helps to get LLVM Bitcode from a project's source code.

REFERENCES

- [1] "Github topics: Mutation testing." [Online]. Available: <https://github.com/topics/mutation-testing>
- [2] H. Coles, "Pitest." [Online]. Available: <http://pitest.org>
- [3] M. Schirp, "Mutant." [Online]. Available: <https://github.com/mbj/mutant>
- [4] C. Lattner and V. Adve, "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*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: <http://dl.acm.org/citation.cfm?id=977395.977673>
- [5] A. Denisov and S. Pankevich, "Mull." [Online]. Available: <https://github.com/mull-project/mull>
- [6] "Apache license," Apache Software Foundation. [Online]. Available: <https://www.apache.org/licenses/LICENSE-2.0>
- [7] "LLVM Language Reference Manual: Instruction Reference." [Online]. Available: <https://releases.llvm.org/3.9.0/docs/LangRef.html#instruction-reference>
- [8] H. Coles, "Pitest: Available mutation operations." [Online]. Available: <http://pitest.org/quickstart/mutators/>
- [9] "RODOS." [Online]. Available: [https://en.wikipedia.org/wiki/Rodos_\(operating_system\)](https://en.wikipedia.org/wiki/Rodos_(operating_system))
- [10] "OpenSSL." [Online]. Available: <https://www.openssl.org>
- [11] "LLVM." [Online]. Available: <https://llvm.org>
- [12] "CppUnit." [Online]. Available: <https://sourceforge.net/projects/cppunit/>
- [13] "LLVM Link Time Optimization: Design and Implementation." [Online]. Available: <https://llvm.org/docs/LinkTimeOptimization.html>
- [14] "GoogleTest." [Online]. Available: <https://github.com/google/googletest>
- [15] "MCJIT TLS support: Cannot select: X86ISD::WrapperRIP." [Online]. Available: https://bugs.llvm.org/show_bug.cgi?id=21431
- [16] H. Coles, "Junk Mutations." [Online]. Available: <https://twitter.com/Ohjc/status/478896988784963584>
- [17] "[llvm-dev] Is it possible to execute Objective-C code via LLVM JIT?" [Online]. Available: <http://lists.llvm.org/pipermail/llvm-dev/2016-October/106218.html>
- [18] P. Delgado-Prez, I. Medina-Bulo, F. Palomo-Lozano, A. Garcia-Domnguez, and J. Domnguez-Jimnez, "Assessment of class mutation operators for c++ with the mucpp mutation system," vol. 81, p. 169184, 01 2017.
- [19] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sept 2011.