Mull

Release 0.7.0

Alex Denisov <alex@lowlevelbits.org>, Stanislav Pankevich <s.pa

Jan 02, 2021

CONTENTS

CHAPTER

ONE

GETTING STARTED

CHAPTER

TWO

INTRODUCTION TO MUTATION TESTING

Mutation Testing is a *fault-based* software testing technique. It evaluates the quality of a test suite by calculating *mutation score* and showing gaps in *semantic coverage*. It does so by creating several slightly modified versions of the original program, *mutants*, and running the test suite against each of them. A mutant is considered to be *killed* if the test suite detects the change, or *survived* otherwise. A mutant is killed if at least one of the tests starts failing.

Each mutation of original program is based on a set of *mutation operators* (or *mutators*). A mutator is a predefined rule that either changes or removes an existing statement or expression in the original program. Each rule is deterministic: the same set of mutation operators applied to the same program results in the same set of mutants.

Mutation score is a ratio of killed vs total mutants. E.g., if seven out of ten mutants are killed, then the score is 0.7, or 70%. The higher the score the better.

CHAPTER

THREE

INSTALLATION

Mull comes with a number of precompiled binaries for macOS and Ubuntu. Please, refer to the Hacking on Mull to build Mull from sources.

3.1 Install on Ubuntu

Get the Bintray public GPG key:

```
wget https://bintray.com/user/downloadSubjectPublicKey?username=bintray -O bintray.key
sudo apt-key add bintray.key
```

Add the repository:

```
sudo echo "deb https://dl.bintray.com/mull-project/ubuntu-18 stable main" >> /etc/apt/
↪sources.list
sudo apt-get update
sudo apt-get install mull
```

Check if everything works:

```
$ mull-cxx --version
Mull: LLVM-based mutation testing
https://github.com/mull-project/mull
Version: 0.6.2
Commit: bd11b48
Date: 13 Mar 2020
LLVM: 9.0.0
```

You can also install Mull for Ubuntu-16.04 or get the latest "nightly" build from the corresponding repositories:

```
deb https://dl.bintray.com/mull-project/ubuntu-16 stable main
deb https://dl.bintray.com/mull-project/ubuntu-18 nightly main
deb https://dl.bintray.com/mull-project/ubuntu-16 nightly main
```

3.2 Install on macOS

Check the latest version on Bintray.

Download and unzip the version you need:

```
wget "https://bintray.com/mull-project/macos/download_file?file_path=Mull-0.6.2-LLVM-
↪9.0-macOS-10.15.3.zip" -O mull.zip
unzip mull.zip
cp ./Mull-0.6.2-LLVM-9.0-macOS-10.15.3/bin/mull-cxx /usr/local/bin/mull-cxx
```

Check the installation:

```
$ mull-cxx --version
Mull: LLVM-based mutation testing
https://github.com/mull-project/mull
Version: 0.6.2
Commit: bd11b48
Date: 13 Mar 2020
LLVM: 9.0.0
```

Installation via Homebrew is on our TODO-list.

You can also get the latest "nightly" build from here.

CHAPTER

FOUR

TUTORIALS

4.1 Hello World Example

The goal of this tutorial is to demonstrate how to run Mull on minimal C programs. After reading it you should have a basic understanding of what arguments Mull needs in order to create mutations in your programs, run the mutants and generate mutation testing reports.

TL;DR version: if you want to run a single copy and paste example, scroll down to `Complete example` below.

The tutorial assumes that you have installed Mull on your system and have the *mull-cxx* executable available:

```
$ mull-cxx -version
Mull: LLVM-based mutation testing
https://github.com/mull-project/mull
Version: 0.5.0
Commit: 791522f
Date: 27 Nov 2019
LLVM: 8.0.0
```

The most important thing that Mull needs to know is the path to your program which must be a valid C or C++ executable. Let's create a C program:

```
int main() {
  return 0;
}
```

and compile it:

```
clang main.cpp -o hello-world
```

We can already try running `mull-cxx` and see what happens:

```
$ mull-cxx hello-world
mull-cxx: for the -test-framework option: must be specified at least once!
```

This is the second important thing that Mull needs: we have to specify which kind of test framework Mull should assume our program uses.

We specify `CustomTest`:

```
mull-cxx -test-framework=CustomTest hello-world
```

`-test-framework=CustomTest` parameter tells Mull that it should not expect a real test framework such as Google Test or any kind of advanced test suite. Instead Mull will simply consider that our tests will be simple test functions which we will call from the `main()` function.

Now the output is different:

```
mull-cxx -test-framework CustomTest hello-world
Extracting bitcode from executable (threads: 1): 0/1No bitcode: x86_64
Extracting bitcode from executable (threads: 1): 1/1. Finished in 1ms.
Loading dynamic libraries (threads: 1): 1/1. Finished in 0ms.
Searching tests (threads: 1): 1/1. Finished in 0ms.
No mutants found. Mutation score: infinitely high

Total execution time: 1ms
```

Notice the `No bitcode: x86_64` warning! Now Mull is already trying to work with our executable but there is still one important detail that is missing: we haven't compiled the program with a special option that embeds LLVM bitcode into our executable.

Mull works on a level of LLVM Bitcode relying on debug information to show results, therefore you should build your project with `-fembed-bitcode` and `-g` flags enabled.

Let's try again:

```
$ clang -fembed-bitcode -g main.cpp -o hello-world
$ mull-cxx -test-framework CustomTest hello-world
Extracting bitcode from executable (threads: 1): 1/1. Finished in 2ms.
Loading bitcode files (threads: 1): 1/1. Finished in 13ms.
Compiling instrumented code (threads: 1): 1/1. Finished in 33ms.
Loading dynamic libraries (threads: 1): 1/1. Finished in 0ms.
Searching tests (threads: 1): 1/1. Finished in 0ms.
Preparing original test run (threads: 1): 1/1. Finished in 2ms.
Running original tests (threads: 1): 1/1. Finished in 10ms.
No mutants found. Mutation score: infinitely high

Total execution time: 62ms
```

The `No bitcode: x86_64` warning has gone and now we can focus on another important part of the output: `No mutants found. Mutation score: infinitely high`. We have our executable but we don't have any code so there is nothing Mull could work on.

Let's add some code:

```
bool valid_age(int age) {
  if (age >= 21) {
    return true;
  }
  return false;
}

int main() {
  int test1 = valid_age(25) == true;
  if (!test1) {
    /// test failed
    return 1;
  }

  int test2 = valid_age(20) == false;
  if (!test2) {
```

(continues on next page)

(continued from previous page)

```
    /// test failed
    return 1;
  }

  /// success
  return 0;
}
```

We compile this new code using the bitcode flags and run the Mull again. This time we also want to add additional flag `-ide-reporter-show-killed` which tells Mull to print killed mutations. Normally we are not interested in seeing killed mutations in console input but in this tutorial we want to be more verbose.

```
$ clang -fembed-bitcode -g main.cpp -o hello-world
$ mull-cxx -test-framework=CustomTest -ide-reporter-show-killed hello-world
Extracting bitcode from executable (threads: 1): 1/1. Finished in 4ms.
Loading bitcode files (threads: 1): 1/1. Finished in 12ms.
Compiling instrumented code (threads: 1): 1/1. Finished in 12ms.
Loading dynamic libraries (threads: 1): 1/1. Finished in 0ms.
Searching tests (threads: 1): 1/1. Finished in 0ms.
Preparing original test run (threads: 1): 1/1. Finished in 1ms.
Running original tests (threads: 1): 1/1. Finished in 12ms.
Applying function filter: no debug info (threads: 1): 1/1. Finished in 10ms.
Applying function filter: file path (threads: 1): 1/1. Finished in 11ms.
Instruction selection (threads: 1): 1/1. Finished in 12ms.
Searching mutants across functions (threads: 1): 1/1. Finished in 10ms.
Applying filter: no debug info (threads: 1): 1/1. Finished in 0ms.
Applying filter: file path (threads: 1): 1/1. Finished in 0ms.
Prepare mutations (threads: 1): 1/1. Finished in 0ms.
Cloning functions for mutation (threads: 1): 1/1. Finished in 13ms.
Removing original functions (threads: 1): 1/1. Finished in 13ms.
Redirect mutated functions (threads: 1): 1/1. Finished in 10ms.
Applying mutations (threads: 1): 1/1. Finished in 12ms.
Compiling original code (threads: 1): 1/1. Finished in 11ms.
Running mutants (threads: 1): 1/1. Finished in 12ms.

Killed mutants (1/2):

/sandbox/mull/tests-lit/tests/tutorials/hello-world/step-5-one-survived-mutations/
↪sample.cpp:13:11: warning: Killed: Replaced >= with < [cxx_ge_to_lt]
  if (age >= 21) {
          ^


Survived mutants (1/2):

/sandbox/mull/tests-lit/tests/tutorials/hello-world/step-5-one-survived-mutations/
↪sample.cpp:13:11: warning: Survived: Replaced >= with > [cxx_ge_to_gt]
  if (age >= 21) {
          ^


Mutation score: 50%

Total execution time: 161ms
```

What we are seeing now is two mutations: one mutation is `Killed`, another one is `Survived`. If we take a closer look at the code and the contents of the tests `test1` and `test2` we will see that one important test case is missing: the one that would test the age `21` and this is exactly what the survived mutation is about: Mull has replaced `age >= 21` with `age > 21` and neither of the two tests have detected the mutation.

Let's add the third test case and see what happens.

4.1.1 Complete example

The code:

```
bool valid_age(int age) {
  if (age >= 21) {
    return true;
  }
  return false;
}

int main() {
  bool test1 = valid_age(25) == true;
  if (!test1) {
    /// test failed
    return 1;
  }

  bool test2 = valid_age(20) == false;
  if (!test2) {
    /// test failed
    return 1;
  }

  bool test3 = valid_age(21) == true;
  if (!test3) {
     /// test failed
     return 1;
  }

  /// success
  return 0;
}
```

```
$ clang -fembed-bitcode -g main.cpp -o hello-world
$ mull-cxx -test-framework=CustomTest -ide-reporter-show-killed hello-world
Extracting bitcode from executable (threads: 1): 1/1. Finished in 2ms.
Loading bitcode files (threads: 1): 1/1. Finished in 12ms.
Compiling instrumented code (threads: 1): 1/1. Finished in 12ms.
Loading dynamic libraries (threads: 1): 1/1. Finished in 0ms.
Searching tests (threads: 1): 1/1. Finished in 0ms.
Preparing original test run (threads: 1): 1/1. Finished in 0ms.
Running original tests (threads: 1): 1/1. Finished in 12ms.
Applying function filter: no debug info (threads: 1): 1/1. Finished in 12ms.
Applying function filter: file path (threads: 1): 1/1. Finished in 13ms.
Instruction selection (threads: 1): 1/1. Finished in 11ms.
Searching mutants across functions (threads: 1): 1/1. Finished in 12ms.
Applying filter: no debug info (threads: 2): 2/2. Finished in 1ms.
Applying filter: file path (threads: 2): 2/2. Finished in 11ms.
Prepare mutations (threads: 1): 1/1. Finished in 0ms.
Cloning functions for mutation (threads: 1): 1/1. Finished in 13ms.
Removing original functions (threads: 1): 1/1. Finished in 10ms.
Redirect mutated functions (threads: 1): 1/1. Finished in 11ms.
Applying mutations (threads: 1): 2/2. Finished in 0ms.
```

(continues on next page)

(continued from previous page)

```
Compiling original code (threads: 1): 1/1. Finished in 11ms.
Running mutants (threads: 2): 2/2. Finished in 12ms.

Killed mutants (2/2):

/sandbox/mull/tests-lit/tests/tutorials/hello-world/step-6-no-survived-mutations/
↪sample.cpp:13:11: warning: Killed: Replaced >= with > [cxx_ge_to_gt]
  if (age >= 21) {
          ^
/sandbox/mull/tests-lit/tests/tutorials/hello-world/step-6-no-survived-mutations/
↪sample.cpp:13:11: warning: Killed: Replaced >= with < [cxx_ge_to_lt]
  if (age >= 21) {
          ^

All mutations have been killed

Mutation score: 100%

Total execution time: 158ms
```

4.1.2 Summary

This is a short summary of what we have learned in tutorial:

- Your code has to be compiled with `-fembed-bitcode -g` compile flags:
  - Mull expects embedded bitcode files to be present in binary executable (ensured by `-fembed-bitcode`).
  - Mull needs debug information to be included by the compiler (enabled by `-g`). Mull uses this information to find mutations in bitcode and source code.
- Mull expects the following arguments to be always provided:
  - Your executable program
  - `-test-framework` parameter that tells Mull which kind of testing framework to expect. In this tutorial we have been using the `CustomTest` framework.

CHAPTER

FIVE

SUPPORTED MUTATION OPERATORS

Operator Name	Operator Semantics
cxx_add_assign_to_sub_assign	Replaces += with -=
cxx_add_to_sub	Replaces + with -
cxx_and_assign_to_or_assign	Replaces &= with |=
cxx_and_to_or	Replaces & with |
cxx_assign_const	Replaces ‘a = b’ with ‘a = 42’
cxx_bitwise_not_to_noop	Replaces ~x with x
cxx_div_assign_to_mul_assign	Replaces /= with *=
cxx_div_to_mul	Replaces / with *
cxx_eq_to_ne	Replaces == with !=
cxx_ge_to_gt	Replaces >= with >
cxx_ge_to_lt	Replaces >= with <
cxx_gt_to_ge	Replaces > with >=
cxx_gt_to_le	Replaces > with <=
cxx_init_const	Replaces ‘T a = b’ with ‘T a = 42’
cxx_le_to_gt	Replaces <= with >
cxx_le_to_lt	Replaces <= with <
cxx_logical_and_to_or	Replaces && with ||
cxx_logical_or_to_and	Replaces || with &&
cxx_lshift_assign_to_rshift_assign	Replaces <<= with >>=
cxx_lshift_to_rshift	Replaces << with >>
cxx_lt_to_ge	Replaces < with >=
cxx_lt_to_le	Replaces < with <=
cxx_minus_to_noop	Replaces -x with x
cxx_mul_assign_to_div_assign	Replaces *= with /=
cxx_mul_to_div	Replaces * with /
cxx_ne_to_eq	Replaces != with ==
cxx_or_assign_to_and_assign	Replaces |= with &=
cxx_or_to_and	Replaces | with &
cxx_post_dec_to_post_inc	Replaces x– with x++
cxx_post_inc_to_post_dec	Replaces x++ with x–
cxx_pre_dec_to_pre_inc	Replaces –x with ++x
cxx_pre_inc_to_pre_dec	Replaces ++x with –x
cxx_rem_assign_to_div_assign	Replaces %= with /=
cxx_rem_to_div	Replaces % with /
cxx_rshift_assign_to_lshift_assign	Replaces >>= with <<=
cxx_rshift_to_lshift	Replaces << with >>
cxx_sub_assign_to_add_assign	Replaces -= with +=

continues on next page

Table 1 – continued from previous page

Operator Name	Operator Semantics
cxx_sub_to_add	Replaces - with +
cxx_xor_assign_to_or_assign	Replaces ^= with |=
cxx_xor_to_or	Replaces ^ with |
negate_mutator	Negates conditionals !x to x and x to !x
remove_void_function_mutator	Removes calls to a function returning void
replace_call_mutator	Replaces call to a function with 42
scalar_value_mutator	Replaces zeros with 42, and non-zeros with 0

CHAPTER

SIX

COMMAND LINE REFERENCE

--workers number How many threads to use

--dry-run Skips real mutants execution. Disabled by default

--cache-dir directory Where to store cache (defaults to /tmp/mull-cache)

--disable-cache Disables cache (enabled by default)

--report-name filename Filename for the report (only for supported reporters). Defaults to <timestamp>.<extension>

--report-dir directory Where to store report (defaults to '.')

--enable-ast Enable "white" AST search (disabled by default)

--reporters reporter Choose reporters:

- **IDE** Prints compiler-like warnings into stdout
- **SQLite** Saves results into an SQLite database
- **Elements** Generates mutation-testing-elements compatible JSON file

--ide-reporter-show-killed Makes IDEReporter to also report killed mutations (disabled by default)

--debug Enables Debug Mode: more logs are printed

--strict Enables Strict Mode: all warning messages are treated as fatal errors

--no-test-output Does not capture output from test runs

--no-mutant-output Does not capture output from mutant runs

--no-output Combines -no-test-output and -no-mutant-output

--compdb-path filename Path to a compilation database (compile_commands.json) for junk detection

--compilation-flags string Extra compilation flags for junk detection

--ld-search-path directory Library search path

--include-path regex File/directory paths to whitelist (supports regex)

--exclude-path regex File/directory paths to ignore (supports regex)

--sandbox sandbox Choose sandbox approach:

- **None** No sandboxing
- **Watchdog** Uses 4 processes, not recommended
- **Timer** Fastest, Recommended

--test-framework framework Choose test framework:

GoogleTest Google Test Framework

CustomTest Custom Test Framework

SimpleTest Simple Test (For internal usage only)

--mutators mutator Choose mutators:

Groups:

all cxx_all, experimental

cxx_all cxx_assignment, cxx_increment, cxx_decrement, cxx_arithmetic, cxx_comparison, cxx_boundary, cxx_bitwise

cxx_arithmetic cxx_minus_to_noop, cxx_add_to_sub, cxx_sub_to_add, cxx_mul_to_div, cxx_div_to_mul, cxx_rem_to_div

cxx_arithmetic_assignment cxx_add_assign_to_sub_assign, cxx_sub_assign_to_add_assign, cxx_mul_assign_to_div_assign, cxx_div_assign_to_mul_assign, cxx_rem_assign_to_div_assign

cxx_assignment cxx_bitwise_assignment, cxx_arithmetic_assignment, cxx_const_assignment

cxx_bitwise cxx_bitwise_not_to_noop, cxx_and_to_or, cxx_or_to_and, cxx_xor_to_or, cxx_lshift_to_rshift, cxx_rshift_to_lshift

cxx_bitwise_assignment cxx_and_assign_to_or_assign, cxx_or_assign_to_and_assign, cxx_xor_assign_to_or_assign, cxx_lshift_assign_to_rshift_assign, cxx_rshift_assign_to_lshift_assign

cxx_boundary cxx_le_to_lt, cxx_lt_to_le, cxx_ge_to_gt, cxx_gt_to_ge

cxx_comparison cxx_eq_to_ne, cxx_ne_to_eq, cxx_le_to_gt, cxx_lt_to_ge, cxx_ge_to_lt, cxx_gt_to_le

cxx_const_assignment cxx_assign_const, cxx_init_const

cxx_decrement cxx_pre_dec_to_pre_inc, cxx_post_dec_to_post_inc

cxx_default cxx_increment, cxx_arithmetic, cxx_comparison, cxx_boundary

cxx_increment cxx_pre_inc_to_pre_dec, cxx_post_inc_to_post_dec

cxx_logical cxx_logical_and_to_or, cxx_logical_or_to_and

experimental negate_mutator, remove_void_function_mutator, scalar_value_mutator, replace_call_mutator, cxx_logical

Single mutators:

cxx_add_assign_to_sub_assign Replaces += with -=

cxx_add_to_sub Replaces + with -

cxx_and_assign_to_or_assign Replaces &= with |=

cxx_and_to_or Replaces & with |

cxx_assign_const Replaces ‘a = b’ with ‘a = 42’

cxx_bitwise_not_to_noop Replaces ~x with x

cxx_div_assign_to_mul_assign Replaces /= with *=

cxx_div_to_mul Replaces / with *

cxx_eq_to_ne Replaces == with !=

cxx_ge_to_gt Replaces >= with >

cxx_ge_to_lt Replaces >= with <

cxx_gt_to_ge Replaces > with >=

cxx_gt_to_le Replaces > with <=

cxx_init_const Replaces ‘T a = b’ with ‘T a = 42’

cxx_le_to_gt Replaces <= with >

cxx_le_to_lt Replaces <= with <

cxx_logical_and_to_or Replaces && with ||

cxx_logical_or_to_and Replaces || with &&

cxx_lshift_assign_to_rshift_assign Replaces <<= with >>=

cxx_lshift_to_rshift Replaces << with >>

cxx_lt_to_ge Replaces < with >=

cxx_lt_to_le Replaces < with <=

cxx_minus_to_noop Replaces -x with x

cxx_mul_assign_to_div_assign Replaces *= with /=

cxx_mul_to_div Replaces * with /

cxx_ne_to_eq Replaces != with ==

cxx_or_assign_to_and_assign Replaces |= with &=

cxx_or_to_and Replaces | with &

cxx_post_dec_to_post_inc Replaces x– with x++

cxx_post_inc_to_post_dec Replaces x++ with x–

cxx_pre_dec_to_pre_inc Replaces –x with ++x

cxx_pre_inc_to_pre_dec Replaces ++x with –x

cxx_rem_assign_to_div_assign Replaces %= with /=

cxx_rem_to_div Replaces % with /

cxx_rshift_assign_to_lshift_assign Replaces >>= with <<=

cxx_rshift_to_lshift Replaces << with >>

cxx_sub_assign_to_add_assign Replaces -= with +=

cxx_sub_to_add Replaces - with +

cxx_xor_assign_to_or_assign Replaces ^= with |=

cxx_xor_to_or Replaces ^ with |

negate_mutator Negates conditionals !x to x and x to !x

remove_void_function_mutator Removes calls to a function returning void

replace_call_mutator Replaces call to a function with 42

scalar_value_mutator Replaces zeros with 42, and non-zeros with 0

CHAPTER

SEVEN

FOR RESEARCHERS

This page contains a short summary of the design and features of Mull. Also the advantages of Mull are highlighted as well as some known issues.

If you want to learn more than we cover here, Mull has a paper: "Mull it over: mutation testing based on LLVM" (see below on this page).

7.1 Design

Mull is based on LLVM and uses its API extensively. The main APIs used are: **LLVM IR**, **LLVM JIT**, **Clang AST API**.

Mull finds and creates mutations of a program in memory, on the level of LLVM bitcode.

Mull uses information about source code obtained via Clang AST API to find which mutations in LLVM bitcode are valid (i.e. they trace back to the source code), all invalid mutations are ignored in a controlled way.

Mull runs the program and its mutated versions in memory using LLVM JIT. The `fork()` call is used to run mutants in child subprocesses so that their execution does not affect Mull as a parent process.

7.2 Mutations search

The default search algorithm simply finds all mutations that can be made on the level of LLVM bitcode.

The **"black search" algorithm** called Junk Detection uses source code information provided by Clang AST to filter out invalid mutations from a set of all possible mutations that are found in LLVM bitcode by the default search algorithm.

The **"white search"** algorithm starts with collecting source code information via Clang AST and then feeds this information to the default search algorithm which allows finding valid mutations and filtering out invalid mutations at the same time.

The black and white search algorithms are very similar in the reasoning that they do. The only difference is that the white search filters out invalid mutations just in time as they are found in LLVM bitcode, while the black search does this after the fact on the raw set of mutations that consists of both valid and invalid mutations.

The black search algorithm appeared earlier and is expected to be more stable. The white search algorithm is currently in development.

7.3 Supported mutation operators

See Supported Mutation Operators.

7.4 Reporting

Mull reports survived/killed mutations to the console by default.

Mull has an SQLite reporter: mutants and execution results are collected in SQLite database. This kind of reporting makes it possible to make SQL queries for a more advanced analysis of mutation results.

Mull supports reporting to HTML via Mutation Testing Elements. Mull generates JSON report which is given to Elements to generate HTML pages.

7.5 Platform support

Mull has a great support of macOS and various Linux systems across all modern versions of LLVM from 3.9.0 to 9.0.0.

Mull supports FreeBSD with minor known issues.

Mull is reported to work on Windows Subsystem for Linux, but no official support yet.

7.6 Test coverage

Mull has 3 layers of testing:

1. Unit and integration testing on the level of C++ classes
2. Integration testing against known real-world projects, such as OpenSSL
3. Integration testing using LLVM Integrated Tester (in progress)

7.7 Current development

The current development goals for Mull for Autumn 2019 - Spring 2020 are:

- Stable performance of black and white search algorithms supported by a solid integration test coverage.
- **Incremental mutation testing**. Mull can already run on subsets of program code but the API and workflows are still evolving.
- More mutation operators.

7.8 Advantages

The main advantage of Mull's design and its approach to finding and doing mutations is very good performance. Combined with incremental mutation testing one can get mutation testing reports in the order of few seconds.

Another advantage is language agnosticism. The developers of Mull have been focusing on C/C++ as their primary development languages at their jobs but the proof of concepts have been developed for the other compiled languages such as Rust and Swift.

A lot of development effort have been put into Mull in order to make it stable across different operating systems and versions of LLVM. Combined with the growing test coverage and highly modular design the authors are slowly but steady getting to the point when they can claim that Mull is a very stable, very well tested and maintained system.

7.9 Known issues

Mull works on the level of LLVM bitcode and from there it gets its strengths but also its main weakness: the precision of the information for mutations is not as high as it is on the source code level. It is a broad area of work where the developers of Mull have to combine the two levels of information about code: LLVM bitcode and AST in order to make Mull both fast and precise. Among other things the good suite of integration tests is aimed to provide Mull with a good contract of supported mutations which are predictable and known to work without any side effects.

7.10 Paper

Mull it over: mutation testing based on LLVM (preprint)

```
@INPROCEEDINGS{8411727,
author={A. Denisov and S. Pankevich},
booktitle={2018 IEEE International Conference on Software Testing, Verification and␣
↪Validation Workshops (ICSTW)},
title={Mull It Over: Mutation Testing Based on LLVM},
year={2018},
volume={},
number={},
pages={25-31},
keywords={just-in-time;program compilers;program testing;program verification;
↪mutations;Mull;LLVM IR;mutated programs;compiled programming languages;LLVM␣
↪framework;LLVM JIT;tested program;mutation testing tool;Testing;Tools;Computer␣
↪languages;Instruments;Runtime;Computer crashes;Open source software;mutation␣
↪testing;llvm},
doi={10.1109/ICSTW.2018.00024},
ISSN={},
month={April},}
```

7.11 Additional information about Mull

- 2019 EuroLLVM Developers' Meeting: A. Denisov "Building an LLVM-based tool: lessons learned" and blog post Building an LLVM-based tool. Lessons learned
- Mutation Testing: implementation details
- Mutation testing for Swift with Mull: how it could work. Looking for contributors
- Mull meets Rust (LLVM Social Berlin #6, 23.02.2017)

CHAPTER

EIGHT

HACKING ON MULL