Mull

Release 0.13.0

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

Nov 23, 2021

CONTENTS

CHAPTER

ONE

GETTING STARTED

Hello there, we are glad to have you here!

If you are new to the subject, then we recommend you start with the little Introduction into Mutation Testing. Then, install Mull and go through the tutorials.

As soon as you are comfortable with the basics you may want to learn about various options and settings Mull has, as well as pick the right set of available mutation operators.

If you are curious about how Mull works under the hood: How Mull works.

If you want to dive deeper and look behind the curtains, then we encourage you to hack on Mull.

If you have any questions feel free to open an issue or join the great community of researchers and practitioners on Discord.

CHAPTER

TWO

FEATURES

- Mull enables mutation testing of C and C++ projects.
- Mull expects a passing test suite to exit with 0. If the test suite fails, it must exit with a non-zero exit code. Any C/C++ test framework that follows this convention is supported by Mull.
- Supported Mutations.
- Generate results in various formats:
  - IDE Reporter: compiler-like warnings are printed to standard output
    - `sample.cpp:15:11: warning: Killed: Replaced >= with < [cxx_ge_to_lt]`
  - SQLite Reporter: SQLite database file.
  - JSON file that conforms mutation-testing-elements schema
  - Mutation Testing Elements HTML Reporter
- Parallelized execution of tests
- Incremental mutation testing. Working with mutations found in Git Diff changesets.
- Mull requires test programs to be compiled with Clang/LLVM. Mull supports all LLVM versions starting from LLVM 6.

For a more detailed description of Mull's architecture, see How Mull works.

CHAPTER

THREE

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

FOUR

INSTALLATION

Mull comes with a number of precompiled binaries for macOS and Ubuntu. There are two flavors of packages:

- stable - tagged releases (0.8.0, 0.9.0, etc.)
- nightly - built for each PR

Alternatively, you can find packages on Github.

Please, refer to the Hacking on Mull to build Mull from sources.

4.1 Install on Ubuntu

Mull supports Ubuntu 18.04 and 20.04.

Setup apt-repository:

```
curl -1sLf 'https://dl.cloudsmith.io/public/mull-project/mull-stable/setup.deb.sh' |␣
↪sudo -E bash
```

Note: Mull uses Cloudsmith for package distribution. The above script detects your OS and sets up the apt repo automagically.

Install the package:

```
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: 0.9.0
Commit: 9f2d43c
Date: 07 Jan 2021
LLVM: 11.0.0
```

You can also get the latest "nightly" build using the corresponding source:

```
curl -1sLf 'https://dl.cloudsmith.io/public/mull-project/mull-nightly/setup.deb.sh' |␣
↪sudo -E bash
```

Links:

- Mull Stable
- Mull Nightly

4.2 Install on macOS

Get the latest version here Github Releases.

Or install via Homebrew:

```
brew install mull-project/mull/mull-stable
```

Check the installation:

```
$ mull-cxx --version
Mull: LLVM-based mutation testing
https://github.com/mull-project/mull
Version: 0.9.0
Commit: 9f2d43c
Date: 07 Jan 2020
LLVM: 11.0.0
```

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

CHAPTER FIVE

TUTORIALS

5.1 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 `Killing mutants again, all killed` below.

5.1.1 Step 1: Checking version

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: 0.8.0
Commit: f94f38ed
Date: 04 Jan 2021
LLVM: 9.0.0
```

5.1.2 Step 2: Enabling Bitcode

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
[info] Extracting bitcode from executable (threads: 1)
[warning] No bitcode: x86_64
       [################################] 1/1. Finished in 3ms
[info] Sanity check run (threads: 1)
       [################################] 1/1. Finished in 409ms
[info] Gathering functions under test (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] No mutants found. Mutation score: infinitely high
[info] Total execution time: 413ms
```

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 hello-world
[info] Extracting bitcode from executable (threads: 1)
       [################################] 1/1. Finished in 5ms
[info] Loading bitcode files (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Sanity check run (threads: 1)
       [################################] 1/1. Finished in 336ms
[info] Gathering functions under test (threads: 1)
       [################################] 1/1. Finished in 1ms
[info] Applying function filter: no debug info (threads: 1)
       [################################] 1/1. Finished in 10ms
[info] Applying function filter: file path (threads: 1)
       [################################] 1/1. Finished in 10ms
[info] Instruction selection (threads: 1)
       [################################] 1/1. Finished in 13ms
[info] Searching mutants across functions (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] No mutants found. Mutation score: infinitely high
[info] Total execution time: 400ms
```

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.

5.1.3 Step 3: Killing mutants, one survived

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) {
    /// 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 output but in this tutorial we want to be more verbose.

```
$ clang -fembed-bitcode -g main.cpp -o hello-world
$ mull-cxx -ide-reporter-show-killed hello-world
[info] Extracting bitcode from executable (threads: 1)
       [################################] 1/1. Finished in 6ms
[info] Loading bitcode files (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Sanity check run (threads: 1)
       [################################] 1/1. Finished in 341ms
[info] Gathering functions under test (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Applying function filter: no debug info (threads: 3)
       [################################] 3/3. Finished in 0ms
[info] Applying function filter: file path (threads: 2)
       [################################] 2/2. Finished in 0ms
[info] Instruction selection (threads: 2)
       [################################] 2/2. Finished in 11ms
[info] Searching mutants across functions (threads: 2)
       [################################] 2/2. Finished in 10ms
[info] Applying filter: no debug info (threads: 6)
       [################################] 6/6. Finished in 1ms
[info] Applying filter: file path (threads: 6)
       [################################] 6/6. Finished in 0ms
```

(continues on next page)

(continued from previous page)

```
[info] Applying filter: junk (threads: 6)
       [################################] 6/6. Finished in 11ms
[info] Prepare mutations (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Cloning functions for mutation (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Removing original functions (threads: 1)
       [################################] 1/1. Finished in 10ms
[info] Redirect mutated functions (threads: 1)
       [################################] 1/1. Finished in 10ms
[info] Applying mutations (threads: 1)
       [################################] 4/4. Finished in 12ms
[info] Compiling original code (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Link mutated program (threads: 1)
       [################################] 1/1. Finished in 109ms
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 360ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 18ms
[info] Running mutants (threads: 4)
       [################################] 4/4. Finished in 63ms
[info] Killed mutants (3/4):
/tmp/sc-PzmaCNIRu/main.cpp:2:15: warning: Killed: Replaced >= with < [cxx_ge_to_lt]
      if (age >= 21) {
              ^
/tmp/sc-PzmaCNIRu/main.cpp:9:33: warning: Killed: Replaced == with != [cxx_eq_to_ne]
      int test1 = valid_age(25) == true;
                                ^
/tmp/sc-PzmaCNIRu/main.cpp:15:33: warning: Killed: Replaced == with != [cxx_eq_to_ne]
      int test2 = valid_age(20) == false;
                                ^
[info] Survived mutants (1/4):
/tmp/sc-PzmaCNIRu/main.cpp:2:15: warning: Survived: Replaced >= with > [cxx_ge_to_gt]
      if (age >= 21) {
              ^
[info] Mutation score: 75%
[info] Total execution time: 996ms
```

What we are seeing now is four mutations: three mutations are `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.

5.1.4 Step 4: Killing mutants again, all killed

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 -ide-reporter-show-killed hello-world
[info] Extracting bitcode from executable (threads: 1)
       [###################################] 1/1. Finished in 4ms
[info] Loading bitcode files (threads: 1)
       [###################################] 1/1. Finished in 11ms
[info] Sanity check run (threads: 1)
       [###################################] 1/1. Finished in 7ms
[info] Gathering functions under test (threads: 1)
       [###################################] 1/1. Finished in 0ms
[info] Applying function filter: no debug info (threads: 3)
       [###################################] 3/3. Finished in 0ms
[info] Applying function filter: file path (threads: 2)
       [###################################] 2/2. Finished in 0ms
[info] Instruction selection (threads: 2)
       [###################################] 2/2. Finished in 12ms
[info] Searching mutants across functions (threads: 2)
       [###################################] 2/2. Finished in 10ms
[info] Applying filter: no debug info (threads: 5)
```

(continues on next page)

(continued from previous page)

```
       [################################] 5/5. Finished in 0ms
[info] Applying filter: file path (threads: 5)
       [################################] 5/5. Finished in 1ms
[info] Applying filter: junk (threads: 5)
       [################################] 5/5. Finished in 12ms
[info] Prepare mutations (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Cloning functions for mutation (threads: 1)
       [################################] 1/1. Finished in 10ms
[info] Removing original functions (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Redirect mutated functions (threads: 1)
       [################################] 1/1. Finished in 10ms
[info] Applying mutations (threads: 1)
       [################################] 5/5. Finished in 0ms
[info] Compiling original code (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Link mutated program (threads: 1)
       [################################] 1/1. Finished in 62ms
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 311ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 19ms
[info] Running mutants (threads: 5)
       [################################] 5/5. Finished in 63ms
[info] Killed mutants (5/5):
/tmp/sc-PzmaCNIRu/main.cpp:2:15: warning: Killed: Replaced >= with > [cxx_ge_to_gt]
      if (age >= 21) {
              ^
/tmp/sc-PzmaCNIRu/main.cpp:2:15: warning: Killed: Replaced >= with < [cxx_ge_to_lt]
      if (age >= 21) {
              ^
/tmp/sc-PzmaCNIRu/main.cpp:9:34: warning: Killed: Replaced == with != [cxx_eq_to_ne]
      bool test1 = valid_age(25) == true;
                                 ^
/tmp/sc-PzmaCNIRu/main.cpp:15:34: warning: Killed: Replaced == with != [cxx_eq_to_ne]
      bool test2 = valid_age(20) == false;
                                 ^
/tmp/sc-PzmaCNIRu/main.cpp:21:34: warning: Killed: Replaced == with != [cxx_eq_to_ne]
      bool test3 = valid_age(21) == true;
                                 ^
[info] All mutations have been killed
[info] Mutation score: 100%
[info] Total execution time: 554ms
```

In this last run, we see that all mutants were killed since we covered with tests all cases around the <=.

5.1.5 Summary

This is a short summary of what we have learned in the tutorial. Your code has to be compiled with `-fembed-bitcode -g` compile flags:

- Mull expects embedded bitcode files to be present in a 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.

The next step is to learn about Compilation Database and Junk Mutations

5.2 Compilation Database and Junk Mutations

This tutorial shows how to apply Mull on a real-world project and how to overcome typical issues you might face.

This tutorial uses fmtlib as an example.

Get sources and build fmtlib:

```
git clone https://github.com/fmtlib/fmt.git
cd fmt
mkdir build.dir
cd build.dir
cmake \
  -DCMAKE_CXX_FLAGS="-fembed-bitcode -g -O0" \
  -DCMAKE_BUILD_TYPE=Debug \
  -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..
make core-test
```

Note: last tested against commit 018688da2a58ba25cdf173bd899734f755adb11a

Run Mull against the `core-test`:

```
mull-cxx -mutators=cxx_add_to_sub ./bin/core-test
```

Right now you should see a weird and long error message by the end of execution. Here is a snippet:

```
/// skipped
[info] Applying mutations (threads: 1)
       [################################] 10/10. Finished in 11ms
[info] Compiling original code (threads: 4)
       [################################] 4/4. Finished in 5671ms
[info] Link mutated program (threads: 1)
[error] Cannot link program
status: Failed
time: 20096ms
exit: -60
command: clang /var/folders/1s/hps840w156xfn_m__h2m17880000gp/T/mull-b5963a.o /var/
↪folders/1s/hps840w156xfn_m__h2m17880000gp/T/mull-62252e.o /var/folders/1s/
↪hps840w156xfn_m__h2m17880000gp/T/mull-22ed08.o /var/folders/1s/hps840w156xfn_m__
↪h2m17880000gp/T/mull-84dd4a.o -o /var/folders/1s/hps840w156xfn_m__h2m17880000gp/T/mull-
↪84a88a.exe
stdout:
stderr: Undefined symbols for architecture x86_64:
```

(continues on next page)

(continued from previous page)

```
  "std::logic_error::what() const", referenced from:
      vtable for assertion_failure in mull-b5963a.o
  "std::runtime_error::what() const", referenced from:
      vtable for testing::internal::GoogleTestFailureException in mull-22ed08.o
      vtable for fmt::v7::format_error in mull-84dd4a.o
      vtable for fmt::v7::system_error in mull-84dd4a.o
/// skipped
```

In order to do the job, Mull takes the executable, deconstructs it into a number of pieces, inserts mutations into those pieces, and then constructs the executable again. In order to re-build the mutated program, Mull needs a linker. As a safe default, it just uses `clang` which works in most of the cases. However, in this case we deal with C++ which needs a corresponding C++ linker. Instead we should be using `clang++`, which will do the job just fine.

Note: on Linux you may have to specify clang-<version> or clang++-<version>, where <version> corresponds to the version of clang installed

Try this:

```
mull-cxx --linker=clang++ -mutators=cxx_add_to_sub ./bin/core-test
```

You should see similar output:

```
[info] Extracting bitcode from executable (threads: 1)
       [#################################] 1/1. Finished in 194ms
[info] Loading bitcode files (threads: 4)
       [#################################] 4/4. Finished in 484ms
[info] Sanity check run (threads: 1)
       [#################################] 1/1. Finished in 12ms
[info] Gathering functions under test (threads: 1)
       [#################################] 1/1. Finished in 5ms
[info] Applying function filter: no debug info (threads: 8)
       [#################################] 12812/12812. Finished in 26ms
[info] Applying function filter: file path (threads: 8)
       [#################################] 12418/12418. Finished in 68ms
[info] Instruction selection (threads: 8)
       [#################################] 12418/12418. Finished in 291ms
[info] Searching mutants across functions (threads: 8)
       [#################################] 12418/12418. Finished in 42ms
[info] Applying filter: no debug info (threads: 8)
       [#################################] 863/863. Finished in 1ms
[info] Applying filter: file path (threads: 8)
       [#################################] 863/863. Finished in 11ms
[info] Applying filter: junk (threads: 8)
/tmp/sc-0Puh0WBoL/fmt/test/./gmock-gtest-all.cc:39:10: fatal error: 'gtest.h' file not␣
↪found
#include "gtest.h"
         ^~~~~~~~~
[warning] Cannot parse file: '/tmp/sc-0Puh0WBoL/fmt/test/./gmock-gtest-all.cc':
mull-cxx /tmp/sc-0Puh0WBoL/fmt/test/./gmock-gtest-all.cc
Make sure that the flags provided to Mull are the same flags that are used for normal␣
↪compilation.
/tmp/sc-0Puh0WBoL/fmt/test/./core-test.cc:8:10: fatal error: 'algorithm' file not found
#include <algorithm>
```

(continues on next page)

(continued from previous page)

```
         ^~~~~~~~~~~
       [--------------------------------] 1/863
[warning] Cannot parse file: '/tmp/sc-0Puh0WBoL/fmt/test/./core-test.cc':
mull-cxx /tmp/sc-0Puh0WBoL/fmt/test/./core-test.cc
Make sure that the flags provided to Mull are the same flags that are used for normal␣
↪compilation.
/tmp/sc-0Puh0WBoL/fmt/src/format.cc:8:10: fatal error: 'fmt/format-inl.h' file not found
#include "fmt/format-inl.h"
         ^~~~~~~~~~~~~~~~~~
[warning] Cannot parse file: '/tmp/sc-0Puh0WBoL/fmt/src/format.cc':
mull-cxx /tmp/sc-0Puh0WBoL/fmt/src/format.cc
Make sure that the flags provided to Mull are the same flags that are used for normal␣
↪compilation.
       [################################] 863/863. Finished in 160ms
[info] Prepare mutations (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Cloning functions for mutation (threads: 4)
       [################################] 4/4. Finished in 51ms
[info] Removing original functions (threads: 4)
       [################################] 4/4. Finished in 43ms
[info] Redirect mutated functions (threads: 4)
       [################################] 4/4. Finished in 12ms
[info] Applying mutations (threads: 1)
       [################################] 10/10. Finished in 10ms
[info] Compiling original code (threads: 4)
       [################################] 4/4. Finished in 5623ms
[info] Link mutated program (threads: 1)
       [################################] 1/1. Finished in 402ms
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 597ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 30ms
[info] Running mutants (threads: 8)
       [################################] 10/10. Finished in 157ms
[info] Survived mutants (10/10):
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:2922:18: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    } else if (i + 1 < num_chars && IsUtf16SurrogatePair(str[i], str[i + 1])) {
                 ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:2922:72: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    } else if (i + 1 < num_chars && IsUtf16SurrogatePair(str[i], str[i + 1])) {
                                                                       ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:554:67: warning: Survived: Replaced + with␣
↪- [cxx_add_to_sub]
                       static_cast<unsigned int>(kMaxRandomSeed)) +
                                                                  ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:566:30: warning: Survived: Replaced + with␣
↪- [cxx_add_to_sub]
  const int next_seed = seed + 1;
                             ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:734:37: warning: Survived: Replaced + with␣
↪- [cxx_add_to_sub]
```

(continues on next page)

(continued from previous page)

```
    const int last_in_range = begin + range_width - 1;
                                    ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:6283:26: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        argv[j] = argv[j + 1];
                         ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:6283:26: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        argv[j] = argv[j + 1];
                         ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:9763:53: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    const int actual_to_skip = stack_frames_to_skip + 1;
                                                    ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:11208:26: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        argv[j] = argv[j + 1];
                         ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:11208:26: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        argv[j] = argv[j + 1];
                         ^
[info] Mutation score: 0%
[info] Total execution time: 8252ms
```

Almost everything works fine, except of those weird warnings:

```
/tmp/sc-0Puh0WBoL/fmt/src/format.cc:8:10: fatal error: 'fmt/format-inl.h' file not found
#include "fmt/format-inl.h"
         ^~~~~~~~~~~~~~~~~~
[warning] Cannot parse file: '/tmp/sc-0Puh0WBoL/fmt/src/format.cc':
mull-cxx /tmp/sc-0Puh0WBoL/fmt/src/format.cc
Make sure that the flags provided to Mull are the same flags that are used for normal␣
↪compilation.
```

That is because of junk mutations.

5.2.1 Junk Mutations

Not every mutation found at Bitcode level can be represented at the source level. A mutation is called *junk mutation* if it exists on the bitcode level, but cannot be achieved on the source code level. Mull filters them out by looking back at the source code. It tries its best, but sometimes it cannot parse the file because it doesn't have enough information. To give all the information needed you should provide compilation database, or compilation flags, or both.

Please, note: Clang adds implicit header search paths, which must be added explicitly via `-compilation-flags`. You can get them using the following commands, for C and C++ respectively:

```
> clang -x c -c /dev/null -v
... skipped
#include <...> search starts here:
 /usr/local/include
 /opt/llvm/10.0.0/lib/clang/10.0.0/include
```

(continues on next page)

(continued from previous page)

```
 /System/Library/Frameworks (framework directory)
 /Library/Frameworks (framework directory)
End of search list.
```

```
> clang++ -x c++ -c /dev/null -v
#include <...> search starts here:
 /opt/llvm/10.0.0/bin/../include/c++/v1
 /usr/local/include
 /opt/llvm/10.0.0/lib/clang/10.0.0/include
 /System/Library/Frameworks (framework directory)
 /Library/Frameworks (framework directory)
End of search list.
```

The paths on your machine might be different, but based on the output above you need the following include dirs:

```
/opt/llvm/10.0.0/include/c++/v1
/usr/local/include
/opt/llvm/10.0.0/lib/clang/10.0.0/include
/usr/include
```

Here is how you can run Mull with junk detection enabled:

```
mull-cxx \
  -linker=clang++ \
  -mutators=cxx_add_to_sub \
  -compdb-path compile_commands.json \
  -compilation-flags="\
    -isystem /opt/llvm/10.0.0/include/c++/v1 \
    -isystem /opt/llvm/10.0.0/lib/clang/10.0.0/include \
    -isystem /usr/include \
    -isystem /usr/local/include" \
    ./bin/core-test
```

You should see similar output:

```
[info] Extracting bitcode from executable (threads: 1)
       [################################] 1/1. Finished in 182ms
[info] Loading bitcode files (threads: 4)
       [################################] 4/4. Finished in 409ms
[info] Sanity check run (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Gathering functions under test (threads: 1)
       [################################] 1/1. Finished in 5ms
[info] Applying function filter: no debug info (threads: 8)
       [################################] 12812/12812. Finished in 22ms
[info] Applying function filter: file path (threads: 8)
       [################################] 12418/12418. Finished in 71ms
[info] Instruction selection (threads: 8)
       [################################] 12418/12418. Finished in 270ms
[info] Searching mutants across functions (threads: 8)
       [################################] 12418/12418. Finished in 43ms
[info] Applying filter: no debug info (threads: 8)
```

(continues on next page)

(continued from previous page)

```
       [#################################] 863/863. Finished in 12ms
[info] Applying filter: file path (threads: 8)
       [#################################] 863/863. Finished in 10ms
[info] Applying filter: junk (threads: 8)
       [#################################] 863/863. Finished in 4531ms
[info] Prepare mutations (threads: 1)
       [#################################] 1/1. Finished in 1ms
[info] Cloning functions for mutation (threads: 4)
       [#################################] 4/4. Finished in 439ms
[info] Removing original functions (threads: 4)
       [#################################] 4/4. Finished in 241ms
[info] Redirect mutated functions (threads: 4)
       [#################################] 4/4. Finished in 12ms
[info] Applying mutations (threads: 1)
       [#################################] 350/350. Finished in 11ms
[info] Compiling original code (threads: 4)
       [#################################] 4/4. Finished in 4570ms
[info] Link mutated program (threads: 1)
       [#################################] 1/1. Finished in 292ms
[info] Warm up run (threads: 1)
       [#################################] 1/1. Finished in 614ms
[info] Baseline run (threads: 1)
       [#################################] 1/1. Finished in 30ms
[info] Running mutants (threads: 8)
       [#################################] 350/350. Finished in 4421ms
[info] Survived mutants (305/350):
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:1758:34: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  state_ = (1103515245U * state_ + 12345U) % kMaxRange;
                                 ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:2275:55: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  return static_cast<TimeInMillis>(now.tv_sec) * 1000 + now.tv_usec / 1000;
                                                      ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:2922:18: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    } else if (i + 1 < num_chars && IsUtf16SurrogatePair(str[i], str[i + 1])) {
                 ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:2922:72: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    } else if (i + 1 < num_chars && IsUtf16SurrogatePair(str[i], str[i + 1])) {
                                                                       ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:2924:63: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
          CreateCodePointFromUtf16SurrogatePair(str[i], str[i + 1]);
                                                              ^


/// skipped

/tmp/sc-0Puh0WBoL/fmt/include/fmt/format-inl.h:1334:68: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  int num_bigits() const { return static_cast<int>(bigits_.size()) + exp_; }
```

(continues on next page)

(continued from previous page)

```
                                                                   ^
/tmp/sc-0Puh0WBoL/fmt/include/fmt/format-inl.h:1284:53: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
      double_bigit result = bigits_[i] * wide_value + carry;
                                                    ^
[info] Mutation score: 12%
[info] Total execution time: 16280ms
```

In the end, 305 out of 350 mutants survived. Why so? One of the reasons is because most of the mutants are unreachable by the test suite. You can learn how to handle this issue in the next tutorial: Keeping mutants under control

5.3 Keeping mutants under control

This tutorial shows you how to keep the number of mutants under control. It builds on top of the Compilation Database and Junk Mutations tutorial so make sure you go through that one first.

When you apply mutation testing for the first time, you might be overwhelmed by the number of mutants - what do you do when you see that several hundred or thousands of mutants survived?

The right way to go about it is to put the number of mutants under control and work through them incrementally.

5.3.1 Mutation Operators

If you apply Mull with the default set of mutation operators on fmtlib, you will get around ~4000 mutants, ~3300 of which survive.

```
$ mull-cxx \
    -linker=clang++ \
    -compdb-path compile_commands.json \
    -compilation-flags="\
      -isystem /opt/llvm/10.0.0/include/c++/v1 \
      -isystem /opt/llvm/10.0.0/lib/clang/10.0.0/include \
      -isystem /usr/include \
      -isystem /usr/local/include" \
    ./bin/core-test

/// skipped

[info] Survived mutants (3397/3946):
/tmp/sc-0Puh0WBoL/fmt/include/fmt/core.h:734:19: warning: Survived: Replaced <= with <␣
↪[cxx_le_to_lt]
    size_ = count <= capacity_ ? count : capacity_;
                  ^
/tmp/sc-0Puh0WBoL/fmt/include/fmt/core.h:1716:12: warning: Survived: Replaced >= with >␣
↪[cxx_ge_to_gt]
    if (id >= detail::max_packed_args) return arg;
           ^
/// skipped

/tmp/sc-0Puh0WBoL/fmt/include/fmt/format-inl.h:1283:51: warning: Survived: Replaced ++x␣
↪with --x [cxx_pre_inc_to_pre_dec]
```

(continues on next page)

(continued from previous page)

```
    for (size_t i = 0, n = bigits_.size(); i < n; ++i) {
                                                ^
[info] Mutation score: 13%
[info] Total execution time: 89999ms
```

Going through all of them to see which ones deserve your attention is simply impractical.

The easiest way to decrease this number is to pick one or two mutation operators.

Let's see how the numbers change if we only use `cxx_add_to_sub` that replaces all the `a + b` to `a - b`.

```
$ mull-cxx \
    -linker=clang++ \
    -mutators=cxx_add_to_sub \
    -compdb-path compile_commands.json \
    -compilation-flags="\
      -isystem /opt/llvm/10.0.0/include/c++/v1 \
      -isystem /opt/llvm/10.0.0/lib/clang/10.0.0/include \
      -isystem /usr/include \
      -isystem /usr/local/include" \
    ./bin/core-test

/// skipped

[info] Survived mutants (305/350):
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:1758:34: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  state_ = (1103515245U * state_ + 12345U) % kMaxRange;
                                 ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:2275:55: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  return static_cast<TimeInMillis>(now.tv_sec) * 1000 + now.tv_usec / 1000;
                                                      ^
/// skipped
/tmp/sc-0Puh0WBoL/fmt/include/fmt/format-inl.h:1334:68: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  int num_bigits() const { return static_cast<int>(bigits_.size()) + exp_; }
                                                                   ^
/tmp/sc-0Puh0WBoL/fmt/include/fmt/format-inl.h:1284:53: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
      double_bigit result = bigits_[i] * wide_value + carry;
                                                    ^
[info] Mutation score: 12%
[info] Total execution time: 18481ms
```

You are still getting plenty - 305 survived out of 350 total, but this is much more manageable.

5.3.2 Filters

You may notice that the last run had, among others, the following mutants survived:

```
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:1758:34: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  state_ = (1103515245U * state_ + 12345U) % kMaxRange;
                                 ^
/tmp/sc-0Puh0WBoL/fmt/test/gmock-gtest-all.cc:2275:55: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  return static_cast<TimeInMillis>(now.tv_sec) * 1000 + now.tv_usec / 1000;
                                                      ^
```

Looking at the paths, it is clear that these mutants are part of the GoogleTest framework (`gmock-gtest-all.cc`). It is very unlikely that you are interested in seeing these in the result. Mull comes with two path-based filters `--exclude-path` and `--include-path`. You can use these to either exclude or include mutations based on their file-system location. Let's exclude everything related to GoogleTest:

```
$ mull-cxx \
    -linker=clang++ \
    -mutators=cxx_add_to_sub \
    -exclude-path=".*gtest.*" \
    -exclude-path=".*gmock.*" \
    -compdb-path compile_commands.json \
    -compilation-flags="\
      -isystem /opt/llvm/10.0.0/include/c++/v1 \
      -isystem /opt/llvm/10.0.0/lib/clang/10.0.0/include \
      -isystem /usr/include \
      -isystem /usr/local/include" \
    ./bin/core-test
/// skipped

[info] Survived mutants (275/320):
/tmp/sc-0Puh0WBoL/fmt/include/fmt/format-inl.h:228:35: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  return i >= 0 ? i * char_digits + count_digits<4, unsigned>(n.value[i]) : 1;
                                  ^
```

275/320 vs. 305/350. Better, but still too much.

5.3.3 Code Coverage

In fact, many of the survived mutants can never be detected by the test suite because they are not reachable by any of the tests. We can leverage code coverage information to cut off all those mutants.

For that to work, we need to gather the coverage info first.

```
$ cmake \
    -DCMAKE_CXX_FLAGS="-fembed-bitcode -g -O0 -fprofile-instr-generate -fcoverage-mapping
↪" \
    -DCMAKE_BUILD_TYPE=Debug \
    -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..
$ make core-test
$ ./bin/core-test
```

Running `core-test` with the coverage info enabled (`-fprofile-instr-generate -fcoverage-mapping`) generates raw coverage info in the current folder. Currently, Mull doesn't work with raw info, so we need to post-process it manually:

```
$ llvm-profdata merge default.profraw -o default.profdata
```

Now we can pass `default.profdata` to Mull. Another important detail, now we also need to tell Mull about additional linker flags - otherwise, it won't be able to reconstruct mutated executable. See the `--linker-flags` CLI option:

```
$ mull-cxx \
    -linker=clang++ \
    -linker-flags="-fprofile-instr-generate -fcoverage-mapping" \
    -mutators=cxx_add_to_sub \
    -exclude-path=".*gtest.*" \
    -exclude-path=".*gmock.*" \
    -coverage-info=default.profdata \
    -compdb-path compile_commands.json \
    -compilation-flags="\
      -isystem /opt/llvm/10.0.0/include/c++/v1 \
      -isystem /opt/llvm/10.0.0/lib/clang/10.0.0/include \
      -isystem /usr/include \
      -isystem /usr/local/include" \
    ./bin/core-test
/// skipped

[info] Survived mutants (14/27):
/tmp/sc-0Puh0WBoL/fmt/include/fmt/format-inl.h:2129:37: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  const int beta_minus_1 = exponent + floor_log2_pow10(-minus_k);
                                    ^
/// skipped
/tmp/sc-0Puh0WBoL/fmt/include/fmt/format.h:1570:31: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
  auto it = reserve(out, size + padding * specs.fill.size());
                              ^
[info] Mutation score: 48%
[info] Total execution time: 14124ms
```

Now, we've got only 27 mutants instead of 4000 in the beginning - something we can work with. It's always a good idea to start with the code coverage in the first place. In this case, even without filters and changing the set of mutation operators, we can decrease the number of mutants to something much more actionable.

As an exercise, try to remove `-exclude-path` and `-mutators` options and see how many mutants you get.

(*Spoiler alert: 563*)

5.4 Non-standard test suites

The goal of this tutorial is to demonstrate how to use Mull with 'non-standard' test suites, such as when the test suite is a separate program. The best example is integration tests written in interpreted languages.

5.4.1 Two-step analysis process

The typical process of applying Mull is a one-step action: run *mull-cxx* and wait for the results. Here is what *mull-cxx* does under the hood:

1. Generates a mutated version of the original program
2. Runs all the mutants
3. Generates report(s)

Since version 0.11.0, there is a way to split this process into a two-step action: run *mull-cxx* to generate mutated program, and then run *mull-runner* to assess all the mutants and generate reports.

Given the program from the Hello World Example the following two runs are identical:

One-step process:

```
$ clang -fembed-bitcode -g main.cpp -o hello-world
$ mull-cxx -ide-reporter-show-killed hello-world
```

Two-step process:

```
$ clang -fembed-bitcode -g main.cpp -o hello-world
$ mull-cxx -mutate-only -output=hello-world-mutated hello-world
$ mull-runner -ide-reporter-show-killed hello-world-mutated
```

While this is useful, let's look into a slightly more complex example.

5.4.2 Tests in interpreted languages

Consider the following (absolutely synthetic) program under test:

```
extern int printf(const char *, ...);
extern int strcmp(const char *, const char *);

int test1(int a, int b) {
  return a + b;
}

int test2(int a, int b) {
  return a * b;
}

int main(int argc, char **argv) {
  if (argc == 1) {
    printf("NOT ENOUGH ARGUMENTS\n");
    return 1;
  }
```

(continues on next page)

(continued from previous page)

```
  if (strcmp(argv[1], "first test") == 0) {
    if (test1(2, 5) == 7) {
      printf("first test passed\n");
      return 0;
    } else {
      printf("first test failed\n");
      return 1;
    }
  } else if (strcmp(argv[1], "second test") == 0) {
    if (test2(2, 5) == 10) {
      printf("second test passed\n");
      return 0;
    } else {
      printf("second test failed\n");
      return 1;
    }
  } else {
    printf("INCORRECT TEST NAME %s\n", argv[1]);
    return 1;
  }
  return 0;
}
```

The program accepts a command-line argument, and depending on the value of the argument it either runs one of the tests or exists with an error. Here is an example:

```
$ clang main.c -o test
$ ./test
NOT ENOUGH ARGUMENTS
$ ./test "first test"
first test passed
$ ./test "second test"
second test passed
$ ./test "third test"
INCORRECT TEST NAME third test
```

Running these tests manually is a tedious and error-prone process, so we create a separate test runner:

```
import sys
import subprocess

test_executable = sys.argv[1]

subprocess.run([test_executable, "first test"], check=True)
subprocess.run([test_executable, "second test"], check=True)
```

The script takes the program under test as its argument and runs the tests against that program.

```
$ clang main.c -o test
$ python3 test.py ./test
first test passed
second test passed
```

In this case, simply using *mull-cxx* is not enough: Mull doesn't know how to run the "external" test suite (*test.py*), so we must be using *mull-runner* for this. The process is two-step.

1. Generate mutated executable

```
$ clang -fembed-bitcode -g main.c -o test
$ mull-cxx -mutate-only \
  -mutators=cxx_add_to_sub -mutators=cxx_mul_to_div \
  -output=test.mutated ./test
[info] Mutate-only mode on: Mull will generate mutants, but won't run them
...
[info] Mutated executable: test.mutated
[info] Total execution time: 182ms
```

2. Run analysis using *mull-runner*:

```
$ mull-runner test.mutated -ide-reporter-show-killed \
  -test-program=python3 -- test.py test.mutated
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 398ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 60ms
[info] Running mutants (threads: 2)
       [################################] 2/2. Finished in 76ms
[info] Killed mutants (2/2):
main.c:5:16: warning: Killed: Replaced + with - [cxx_add_to_sub]
      return a + b;
               ^
main.c:9:16: warning: Killed: Replaced * with / [cxx_mul_to_div]
      return a * b;
               ^
[info] All mutations have been killed
[info] Mutation score: 100%
[info] Total execution time: 535ms
```

Note, *test.mutated* appears twice in the arguments list: the first appearance is required for *mull-runner* to extract the mutants generated at the first step. The second appearance is passed to the test program.

5.5 Working with SQLite report

Warning: the data model has changed and most of the tutorial is no longer relevant/applicable. This tutorial will be updated for the next release.

From the very beginning, we didn't want to impose our vision on treating the results of mutation testing. Some people do not care about the mutation score, while others do care, but want to calculate it slightly differently.

To solve this problem, Mull splits execution and reporting into separate phases. What Mull does is apply mutation testing on a program, collect as much information as possible, and then pass this information to one of several reporters.

At the moment of writing, there are three reporters:

- `IDEReporter`: prints mutants in the format of clang warnings
- `MutationTestingElementsReporter`: emits a JSON-file compatible with Mutation Testing Elements.
- `SQLiteReporter`: saves all the information to an SQLite database

One of the ways to do a custom analysis of mutation testing results is to run queries against the SQLite database. The rest of this document describes how to work with Mull's SQLite database.

5.5.1 Database Schema

The following picture describes part of the existing database:

Some fields and tables irrelevant for this document are omitted.

Let's take a brief look at each table.

test

This table contains information about a particular test. A test, from Mull's perspective, is just a function. For UI reporting purposes, Mull records the location of the function.

mutation_point

This is one of the core elements of Mull. The mutation point describes what was changed and where. The `mutator` field stores name of a mutation operator applied at this mutation point. The rest of the fields describe the physical location of the mutation.

execution_result

Execution results are stored separately from mutation points for the following reasons:

- a mutation point might be reachable by more than one test. Therefore Mull runs several tests against one mutation point
- to gather code coverage information Mull runs all the tests one by one without any mutations involved

In other words, `execution_result` describes many-to-many relation between tests and mutations.

Empty `mutation_point_id` indicates that a test was run to gather code coverage information.

The `status` field stores a numerical value as described in the following table:

Numeric value	String Value	Description
1	Failed	test has failed (the exit code does not equal 0)
2	Passed	test has passed (the exit code equals 0)
3	Timedout	test execution took more time than expected
4	Crashed	test program was terminated
5	AbnormalExit	test program exited (some test frameworks call `exit(1)` when test fails)
6	DryRun	test was not run (DryRun mode enabled)
7	FailFast	mutant was killed by another test so this test run can be skipped

5.5.2 Running Queries

The benefit of having results in an SQLite database is that we can run as many queries as we want and to examine the results without re-running Mull, which can be a long-running task.

If you don't have a sample project ready, then it is a good idea to check out the fmtlib tutorial.

To enable SQLite reporter, add `-reporters=SQLite` to the CLI options. It is also recommended to specify the report name via `-report-name`, e.g.:

```
mull-cxx -mutators=cxx_add_to_sub \
  -compdb-path compile_commands.json \
  -compilation-flags="\
    -isystem /opt/llvm/5.0.0/include/c++/v1 \
    -isystem /opt/llvm/5.0.0/lib/clang/5.0.0/include \
    -isystem /usr/include \
    -isystem /usr/local/include" \
  -reporters=SQLite \
  -report-name=tutorial \
  ./bin/core-test
```

In the end, you should see something like this:

```
[info] Results can be found at './tutorial.sqlite'
```

Open the database and enable better formatting (optional):

```
sqlite3 ./tutorial.sqlite
sqlite> .header on
sqlite> .mode column
```

Now you can examine contents of the database:

```
sqlite> .tables
config                mutation_point        mutation_result
execution_result      mutation_point_debug  test

sqlite> .schema execution_result
CREATE TABLE execution_result (
  test_id TEXT,
  mutation_point_id TEXT,
  status INT,
  duration INT,
  stdout TEXT,
  stderr TEXT
);
```

As you can see, the schema for `execution_result` matches the one on the picture above.

Basic exploration

Let’s check how many mutants:

```
sqlite>  select count(*) from mutation_point;
count(*)
----------
35
```

Let’s see some stats on the execution time:

```
sqlite> select avg(duration), max(duration) from execution_result;
avg(duration)     max(duration)
----------------  -------------
5.23497267759563  76
```

Let’s see what’s wrong with that slow test run:

Note: Here, I use several queries to save some screen space. Locally you may combine this into one query just fine.

```
sqlite> select rowid, status, duration from execution_result order by duration desc␣
↪limit 5;
rowid       status      duration
----------  ----------  ----------
73          3           76
54          1           22
55          1           19
179         1           17
5           2           14
sqlite> select test_id from execution_result where rowid = 73;
test_id
-----------------------
FormatDynArgsTest.Basic
sqlite> select mutation_point_id from execution_result where rowid = 73;
mutation_point_id
-----------------------------------------------------------------------------------------
3539da16613cf5da12032f308b293b8f_3539da16613cf5da12032f308b293b8f_478_2_15_cxx_add_to_sub
```

Now, we now the exact test case and exact mutation we can identify their locations in the source code:

```
sqlite> select * from test where unique_id = "BufferTest.Access";
test_name          unique_id          location_file                            location_
↪line
-----------------  -----------------  ---------------------------------------  ----------
↪---
BufferTest.Access  BufferTest.Access  /tmp/sc-UiYEtcmuH/fmt/test/core-test.cc  144

sqlite> select mutator, filename, line_number, column_number from mutation_point
  where unique_id = "3539da16613cf5da12032f308b293b8f_3539da16613cf5da12032f308b293b8f_
↪478_2_15_cxx_add_to_sub";
mutator         filename                                     line_number  column_number
--------------  -------------------------------------------  -----------  -------------
cxx_add_to_sub  /tmp/sc-UiYEtcmuH/fmt/include/fmt/format.h  1746         45
```

Deeper dive

Exploration via SQLite is cool, but let's do some math and calculate the mutation score using SQL.

To calculate mutation score, we will use the following formula: `# of killed mutants / # of all mutants`, where killed means that the status of an `execution_result` is anything but `Passed`.

Counting all the killed mutants is not the most straightforward query, but should still be bearable: select all the mutation points and then narrow down the results by selecting the ones where the execution status does not equal 2 (Passed).

```
sqlite> select mutation_point.unique_id as mutation_point_id from mutation_point
      inner join execution_result on execution_result.mutation_point_id = mutation_point.
↪unique_id
      where execution_result.status <> 2
      group by mutation_point_id;
```

Reusing this query is a bit of a hassle, so it makes sense to create an SQL View which can be used as a normal table:

```
sqlite> create view killed_mutants as
      select mutation_point.unique_id as mutation_point_id from mutation_point
      inner join execution_result on execution_result.mutation_point_id = mutation_point.
↪unique_id
      where execution_result.status <> 2
      group by mutation_point_id;
sqlite> select count(*) from killed_mutants;
count(*)
----------
16
```

With the number of killed mutants in place we can calculate the mutation score:

```
sqlite> select round(
    (select count(*) from killed_mutants) * 1.0 /
    (select count(*) from mutation_point) * 100) as score;
score
----------
46.0
```

Gotchas

One important thing to remember: by default Mull also stores `stderr` and `stdout` of each test run, which can blow up the size of the database by tens on gigabytes.

If you don't need the `stdout/stderr`, then it is recommended to disable it via one of the following options `--no-output`, `--no-test-output`, `--no-mutant-output`.

Alternatively, you can strip this information from the database using this query:

```
begin transaction;
create temporary table t1_backup as select test_id, mutation_point_id, status, duration␣
↪FROM execution_result;
drop table execution_result;
create table execution_result as select * FROM t1_backup;
drop table t1_backup;
```

(continues on next page)

(continued from previous page)

```
commit;
vacuum;
```

CHAPTER

SIX

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_remove_negation	Replaces !a with a
cxx_remove_void_call	Removes calls to a function returning void
cxx_replace_scalar_call	Replaces call to a function with 42

continues on next page

Table 1 – continued from previous page

Operator Name	Operator Semantics
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
scalar_value_mutator	Replaces zeros with 42, and non-zeros with 0

CHAPTER

SEVEN

INCREMENTAL MUTATION TESTING

Normally, Mull looks for mutations in all files of a project. Depending on a project's size, a number of mutations can be very large, so running Mull against all of them might be a rather slow process. Speed aside, an analysis of a large mutation data sets can be very time consuming work to be done by a user.

Incremental mutation testing is a feature that enables running Mull only on the mutations found in Git Diff changesets. Instead of analysing all files and functions, Mull only finds mutations in the source lines that are covered by a particular Git Diff changeset.

Example: if a Git diff is created from a project's Git tree and the diff is only one line, Mull will only find mutations in that line and will skip everything else.

To enable incremental mutation testing, two arguments have to be provided to Mull: `-git-diff-ref=<branch or commit>` and `-git-project-root=<path>` which is a path to a project's Git root path.

An additional debug option `-debug` can be useful for a visualization of how exactly Mull whitelists or blacklists found source lines.

Note: Incremental mutation testing is an experimental feature. Things might go wrong. If you encounter any issues, please report them on the mull/issues tracker.

7.1 Typical use cases

Under the hood, Mull runs `git diff` from a project's root folder. There are at least three reasonable options for using the `-git-diff-ref` argument:

1. `-git-diff-ref=origin/main`

   Mull is run from a branch with a few commits against a main branch such as `main`, `master` or equivalent. This is what you get from your branch when you simply do `git diff origin/master`. This way you can also test your branch if you have Mull running as part of your CI workflow.

2. `-git-diff-ref=.` (unstaged), `-git-diff-ref=HEAD` (unstaged + staged)

   Mull is run against a diff between the "unclean" tree state and your last commit. This use case is useful when you want to check your work-in-progress code with Mull before committing your changes.

3. `-git-diff-ref=COMMIT^!`

   Mull is run against a diff of a specific commit (see also How can I see the changes in a Git commit?). This option should be used with caution because Mull does not perform a `git checkout` to switch to a given commit's state. Mull always stands on top of the existing tree, so if a provided commit has already been overridden by more recent commits, Mull will not produce the results for that earlier commit which can result in a misleading information in the mutation reports. Use this option only if you are sure that no newer commits in your Git tree have touched the file(s) you are interested in.

CHAPTER

EIGHT

COMMAND LINE REFERENCE

8.1 mull-cxx

--output path	output file
--workers number	How many threads to use
--timeout number	Timeout per test run (milliseconds)
--dry-run	Skips mutant execution and generation. Disabled by default
--mutate-only	Skips mutant execution. Unlike -dry-run generates mutants. Disabled 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 '.')
--report-patch-base directory	Create Patches relative to this directory (defaults to git-project-root if available, else absolute path will be used)
--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
	Patches Generates patch file for each mutation
	GithubAnnotations Print GithubAnnotations for mutants
--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
--keep-object-files	Keep temporary object files
--keep-executable	Keep temporary executable file
--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
--disable-junk-detection	Do not remove junk mutations
--compdb-path filename	Path to a compilation database (compile_commands.json) for junk detection

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

--linker string Linker program

--linker-flags string Extra linker flags to produce final executable

--linker-timeout number Timeout for the linking job (milliseconds)

--coverage-info string Path to the coverage info file (LLVM's profdata)

--include-not-covered Include (but do not run) not covered mutants. Disabled by default

--include-path regex File/directory paths to whitelist (supports regex, equivalent to "grep -E")

--exclude-path regex File/directory paths to ignore (supports regex, equivalent to "grep -E")

--git-diff-ref ref Git branch, commit, or tag to run diff against (enables incremental testing)

--git-project-root path Path to project's Git root (used together with -git-diff-ref)

--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_calls

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_calls cxx_remove_void_call, cxx_replace_scalar_call

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, cxx_remove_negation

experimental negate_mutator, scalar_value_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_remove_negation Replaces !a with a

cxx_remove_void_call Removes calls to a function returning void

cxx_replace_scalar_call Replaces call to a function with 42

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

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

8.2 mull-runner

--test-program path Path to a test program

--workers number How many threads to use

--timeout number Timeout per test run (milliseconds)

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

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

--report-patch-base directory Create Patches relative to this directory (defaults to git-project-root if available, else absolute path will be used)

--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
- **Patches** Generates patch file for each mutation
- **GithubAnnotations** Print GithubAnnotations for mutants

--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

CHAPTER

NINE

HOW MULL WORKS

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).

9.1 Design

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

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

All mutations are injected into original program's code. Each injected mutation is hidden under a conditional flag that enables that specific mutation. The resulting program is compiled into a single binary which is run multiple times, one run per mutation. With each run, Mull activates a condition for a corresponding mutation to check how the injection of that particular mutation affects the execution of a test suite.

Mull runs the tested program and its mutated versions in child subprocesses so that the execution of the tested program does not affect Mull running in a parent process.

Note: Mull no longer uses LLVM JIT for execution of mutated programs. See the *Historical note: LLVM JIT deprecation (January 2021)*.

Mull uses information about source code obtained via Clang AST API to find out 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.

9.2 Mutations search

The default search algorithm simply finds all mutations that can be found 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.

9.3 Supported mutation operators

See Supported Mutation Operators.

9.4 Reporting

Mull reports survived/killed mutations to the console by default. The compiler-like warnings are printed to standard output.

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.

9.5 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.

9.6 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 (LIT)

9.7 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 the primary supported languages but the proof of concepts for other compiled languages, such as Rust and Swift, have been developed.

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, Mull is a very stable, well-tested and maintained system.

9.8 Known issue: Precision

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.

9.9 Historical note: LLVM JIT deprecation (January 2021)

The usage of LLVM JIT has been deprecated and all LLVM JIT-related code has been removed from Mull by January 2021.

This issue explains the reasons: PSA: Moving away from JIT.

9.10 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},}
```

9.11 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

TEN

HACKING ON MULL

10.1 Internals

Before you start hacking it may be helpful to get through the second and third sections of this paper: Mull it over: mutation testing based on LLVM from ICST 2018.

10.2 Development Setup using Vagrant

Mull supplies a number of ready to use virtual machines based on VirtualBox.

The machines are managed using Vagrant and Ansible.

Do the following steps to setup a virtual machine:

```
cd infrastructure
vagrant up debian
```

This command will:

- setup a virtual machine
- install required packages (cmake, sqlite3, pkg-config, . . .)
- download precompiled version of LLVM
- build Mull against the LLVM
- run Mull's test suite
- run Mull against OpenSSL and fmtlib as an integration test

Once the machine is up and running you can start hacking over SSH:

```
vagrant ssh debian
```

Within the virtual machine Mull's sources located under `/opt/mull`.

Alternatively, you can setup a remote toolchain within your IDE, if it supports it.

When you are done feel free to drop the virtual machine:

```
vagrant destroy debian
```

You can see the full list of supplied VMs by running this command:

```
vagrant status
```

10.3 Local Development Setup

You can replicate all the steps managed by Vagrant/Ansible manually.

10.3.1 Required packages

Please, look at the corresponding Ansible playbook (`debian-playbook.yaml`, `macos-playbook.yaml`, etc.) for the list of packages required on your OS.

10.3.2 LLVM

You need LLVM to build and debug Mull. You can use any LLVM version between 6.0 and 12.0.

There are several options:

1. Download precompiled version of LLVM from the official website: http://releases.llvm.org/ This is a recommended option. Use it whenever possible. Simply download the tarball and unpack it somewhere.
2. Build LLVM from scratch on your own This option also works. Use it whenever you cannot or do not want to use precompiled version.
3. Ask Mull to build LLVM for you This is recommended only if you need to debug some issue in Mull that requires deep dive into the LLVM itself.

If you are going for an option 2 or 3 - make sure you also include Clang.

10.3.3 Build Mull

Create a build folder and initialize build system:

```
git clone https://github.com/mull-project/mull.git --recursive
cd mull
mkdir build.dir
cd build.dir
cmake -DPATH_TO_LLVM=path/to/llvm ..
make mull-cxx
make mull-tests
```

The `PATH_TO_LLVM` depends on which option you picked in previous section:

1. Path to extracted tarball.
2. Path to a build directory.
3. Path to a source directory.

If you are getting linker errors, then it is very likely related to the C++ ABI. Depending on your OS/setup you may need to tweak the `_GLIBCXX_USE_CXX11_ABI` (0 or 1):

```
cmake -DPATH_TO_LLVM=some-path -DCMAKE_CXX_FLAGS=-D_GLIBCXX_USE_CXX11_ABI=0 ..
```

If the linker error you get is something like `undefined reference to `typeinfo for irm::CmpInstPredicateReplacement'`, try to pass the `-fno-rtti` flag:

```
cmake -DPATH_TO_LLVM=some-path -DCMAKE_CXX_FLAGS=-fno-rtti ..
```