Mull

Release 0.18.0

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

May 05, 2022

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 9.

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-10 # Ubuntu 18.04
sudo apt-get install mull-12 # Ubuntu 20.04
```

Check if everything works:

```
$ mull-runner-10 --version
Mull: Practical mutation testing for C and C++
Home: https://github.com/mull-project/mull
Docs: https://mull.readthedocs.io
Version: 0.15.0
Commit: ab159cd
Date: 20 Jan 2022
LLVM: 10.0.0
```

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

Download the latest version from Github Releases.

Check the installation:

```
$ mull-runner-13 --version
Mull: Practical mutation testing for C and C++
Home: https://github.com/mull-project/mull
Docs: https://mull.readthedocs.io
Version: 0.15.0
Commit: 0252a4cf
Date: 28 Jan 2022
LLVM: 13.0.0
```

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

CHAPTER

FIVE

TUTORIALS

5.1 Hello World Example

Note: We would love to hear from you!

Please, report any issues on GitHub, or bring your questions to the #mull channel on Discord.

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.

Note: Clang 9 or newer is required!

5.1.1 Step 1: Checking version

Mull comes in a form of a compiler plugin and therefore tied to specific versions of Clang and LLVM. As a consequence of that, tools and plugins have a suffix with the actual Clang/LLVM version.

This tutorial assumes that you are using Clang 12 and that you have *installed* Mull on your system and have the `mull-runner-12` executable available:

```
$ mull-runner-12 -version
Mull: LLVM-based mutation testing
https://github.com/mull-project/mull
Version: 0.15.0
Commit: a4be349e
Date: 18 Jan 2022
LLVM: 12.0.1
```

5.1.2 Step 2: Enabling compiler plugin

Let's create a C++ program:

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

and compile it:

```
$ clang-12 main.cpp -o hello-world
```

We can already try using `mull-runner` and see what happens:

```
$ mull-runner-12 ./hello-world
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 5ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 4ms
[info] No mutants found. Mutation score: infinitely high
[info] Total execution time: 10ms
```

Notice the `No mutants found` message! Now, Mull is ready to work with the executable but there are no mutants: we haven't compiled the program with the compiler plugin that embeds mutants into our executable.

Let's fix that! To pass the plugin to Clang, you need to add a few compiler flags.

Note: For Clang 9, 10, and 11 also pass `-O1`, otherwise the plugin won't be called.

Note: `-grecord-command-line` doesn't currently work if you compile several files in one go, e.g. `clang a.c b.c c.c`. In this case, please remove the flag.

```
$ clang-12 -fexperimental-new-pass-manager \
  -fpass-plugin=/usr/local/lib/mull-ir-frontend-12 \
  -g -grecord-command-line \
  main.cpp -o hello-world
[warning] Mull cannot find config (mull.yml). Using some defaults.
```

Notice the warning: Mull needs a config. However, in this tutorial we can ignore the warning and rely on the defaults.

You can learn more about the config *here*.

Let's run `mull-runner` again:

```
$ mull-runner-12 ./hello-world
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 4ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 6ms
[info] No mutants found. Mutation score: infinitely high
[info] Total execution time: 12ms
```

Still no mutants, but this time it is because we don't have any code Mull can mutate.

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

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

  /// success
  return 0;
}
```

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

```
$ clang-12 -fexperimental-new-pass-manager \
        -fpass-plugin=/usr/local/lib/mull-ir-frontend-12 \
        -g -grecord-command-line \
        main.cpp -o hello-world
$ mull-runner-12 -ide-reporter-show-killed hello-world
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 151ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 3ms
[info] Running mutants (threads: 4)
       [################################] 4/4. Finished in 10ms
[info] Killed mutants (3/4):
/tmp/sc-tTV8a84lL/main.cpp:2:11: warning: Killed: Replaced >= with < [cxx_ge_to_lt]
  if (age >= 21) {
          ^
/tmp/sc-tTV8a84lL/main.cpp:9:30: warning: Killed: Replaced == with != [cxx_eq_to_ne]
  bool test1 = valid_age(25) == true;
                             ^
/tmp/sc-tTV8a84lL/main.cpp:15:30: warning: Killed: Replaced == with != [cxx_eq_to_ne]
  bool test2 = valid_age(20) == false;
                             ^
[info] Survived mutants (1/4):
```

(continues on next page)

(continued from previous page)

```
/tmp/sc-tTV8a84lL/main.cpp:2:11: warning: Survived: Replaced >= with > [cxx_ge_to_gt]
  if (age >= 21) {
          ^
[info] Mutation score: 75%
[info] Total execution time: 167ms
```

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-12 -fexperimental-new-pass-manager \
             -fpass-plugin=/usr/local/lib/mull-ir-frontend-12 \
             -g -grecord-command-line \
             main.cpp -o hello-world
```

(continues on next page)

(continued from previous page)

```
$ mull-runner-12 -ide-reporter-show-killed hello-world
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 469ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 4ms
[info] Running mutants (threads: 5)
       [################################] 5/5. Finished in 12ms
[info] Killed mutants (5/5):
/tmp/sc-tTV8a84lL/main.cpp:2:11: warning: Killed: Replaced >= with > [cxx_ge_to_gt]
  if (age >= 21) {
          ^
/tmp/sc-tTV8a84lL/main.cpp:2:11: warning: Killed: Replaced >= with < [cxx_ge_to_lt]
  if (age >= 21) {
          ^
/tmp/sc-tTV8a84lL/main.cpp:9:30: warning: Killed: Replaced == with != [cxx_eq_to_ne]
  bool test1 = valid_age(25) == true;
                             ^
/tmp/sc-tTV8a84lL/main.cpp:15:30: warning: Killed: Replaced == with != [cxx_eq_to_ne]
  bool test2 = valid_age(20) == false;
                             ^
/tmp/sc-tTV8a84lL/main.cpp:21:30: 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: 487ms
```

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

5.1.5 Summary

As a summary, all you need to enable Mull is to add a few compiler flags to the build system and then run `mull-runner` against the resulting executable. Just to recap:

```
$ clang-12 -fexperimental-new-pass-manager \
              -fpass-plugin=/usr/local/lib/mull-ir-frontend-12 \
              -g -grecord-command-line \
              main.cpp -o hello-world
$ mull-runner-12 hello-world
```

5.1.6 Next Steps

Take a look at *makefile* or *CMake* integrations.

5.2 Makefile Integration: OpenSSL

This tutorial demonstrates how to integrate Mull into a custom Makefile-based build system.

We use OpenSSL as an example.

Note: If you are new to Mull, then *Hello World example* is a good starting point.

5.2.1 Step 1. Check out the source code

`openssl-3.0.1` is the latest version we tested.

```
git clone https://github.com/openssl/openssl.git \
  --branch openssl-3.0.1 \
  --depth 1
```

5.2.2 Step 2. Create sample Mull config

Create a file `openssl/mull.yml` with the following contents:

```
mutators:
  - cxx_add_to_sub
```

5.2.3 Step 3. Configure and build OpenSSL

```
cd openssl
export CC=clang-12
./config -O0 -fexperimental-new-pass-manager \
  -fpass-plugin=/usr/lib/mull-ir-frontend-12 \
  -g -grecord-command-line
make build_generated -j
make ./test/bio_enc_test -j
```

5.2.4 Step 4. Run Mull against OpenSSL's tests

```
mull-runner-12 ./test/bio_enc_test
```

You should see similar (and pretty long) output:

```
[info] Using config /tmp/sc-g6cD7gfN4/openssl/mull.yml
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 638ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 281ms
[info] Running mutants (threads: 8)
       [################################] 1606/1606. Finished in 147786ms
[info] Survived mutants (1588/1606):
/tmp/sc-g6cD7gfN4/openssl/apps/lib/opt.c:1126:15: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
        i = 2 + (int)strlen(o->name);
              ^
/tmp/sc-g6cD7gfN4/openssl/apps/lib/opt.c:1128:20: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
            i += 1 + strlen(valtype2param(o));
                   ^
/tmp/sc-g6cD7gfN4/openssl/crypto/aria/aria.c:546:20: warning: Survived: Replaced + with -
↪ [cxx_add_to_sub]
    int Nr = (bits + 256) / 32;
                   ^
/tmp/sc-g6cD7gfN4/openssl/crypto/asn1/a_bitstr.c:62:13: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    ret = 1 + len;
              ^
<truncated>
/tmp/sc-g6cD7gfN4/openssl/test/testutil/format_output.c:282:47: warning: Survived:␣
↪Replaced + with - [cxx_add_to_sub]
    l1 = bn1 == NULL ? 0 : (BN_num_bytes(bn1) + (BN_is_negative(bn1) ? 1 : 0));
                                               ^
/tmp/sc-g6cD7gfN4/openssl/test/testutil/format_output.c:283:47: warning: Survived:␣
↪Replaced + with - [cxx_add_to_sub]
    l2 = bn2 == NULL ? 0 : (BN_num_bytes(bn2) + (BN_is_negative(bn2) ? 1 : 0));
                                               ^
/tmp/sc-g6cD7gfN4/openssl/test/testutil/format_output.c:301:32: warning: Survived:␣
↪Replaced + with - [cxx_add_to_sub]
    len = ((l1 > l2 ? l1 : l2) + bytes - 1) / bytes * bytes;
                               ^
/tmp/sc-g6cD7gfN4/openssl/test/testutil/random.c:24:54: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    test_random_state[pos] += test_random_state[(pos + 28) % 31];
                                                         ^
[info] Mutation score: 1%
[info] Total execution time: 149344ms
```

Mull says that 1588 out of 1606 mutants survived. That's a lot. Why so many and how do we handle this?

The answer is in the next tutorial *Keeping mutants under control*.

5.3 CMake Integration: fmtlib

This tutorial demonstrates how to integrate Mull into a CMake-based project.

We use fmtlib as an example.

Note: If you are new to Mull, then *Hello World example* is a good starting point.

5.3.1 Step 1. Check out the source code

```
git clone https://github.com/fmtlib/fmt --depth 1
```

5.3.2 Step 2. Create sample Mull config

Create a file `fmt/mull.yml` with the following contents:

```
mutators:
  - cxx_add_to_sub
```

5.3.3 Step 3. Configure and build fmtlib

```
mkdir fmt/build.dir
cd fmt/build.dir
export CXX=clang++-12
cmake \
  -DCMAKE_CXX_FLAGS="-O0 -fexperimental-new-pass-manager -fpass-plugin=/usr/lib/mull-ir-
↪frontend-12 -g -grecord-command-line" \
  ..
make core-test -j
```

5.3.4 Step 4. Run Mull against fmtlib tests

```
mull-runner-12 ./bin/core-test
```

You should see similar output:

```
[info] Using config /tmp/sc-ySbkbNvt3/fmt/mull.yml
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Filter mutants (threads: 1)
       [################################] 1/1. Finished in 1ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 8ms
[info] Running mutants (threads: 8)
       [################################] 164/164. Finished in 363ms
[info] Survived mutants (160/164):
```

(continues on next page)

(continued from previous page)

```
/tmp/sc-ySbkbNvt3/fmt/include/fmt/core.h:2237:14: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
  return len + !len;
              ^
/tmp/sc-ySbkbNvt3/fmt/include/fmt/core.h:2267:24: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
    value = value * 10 + unsigned(*p - '0');
                       ^
/tmp/sc-ySbkbNvt3/fmt/include/fmt/core.h:2277:31: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
                 prev * 10ull + unsigned(p[-1] - '0') <= max
                              ^
<truncated>
/tmp/sc-ySbkbNvt3/fmt/test/gtest/gmock-gtest-all.cc:12210:36: warning: Survived:␣
↪Replaced + with - [cxx_add_to_sub]
               IsUTF8TrailByte(s[i + 1]) &&
                                   ^
/tmp/sc-ySbkbNvt3/fmt/test/gtest/gmock-gtest-all.cc:12211:36: warning: Survived:␣
↪Replaced + with - [cxx_add_to_sub]
               IsUTF8TrailByte(s[i + 2]) &&
                                   ^
/tmp/sc-ySbkbNvt3/fmt/test/gtest/gmock-gtest-all.cc:14386:26: warning: Survived:␣
↪Replaced + with - [cxx_add_to_sub]
        argv[j] = argv[j + 1];
                         ^
[info] Mutation score: 2%
[info] Total execution time: 491ms
```

We've got lots of survived mutants.

We can ignore some of them (specifically the ones coming from `gtest` and `gmock`) by extending the config file as follows:

```
mutators:
  - cxx_add_to_sub
excludePaths:
  - .*gtest.*
  - .*gmock.*
```

After rerunning Mull:

```
$ mull-runner-12 ./bin/core-test
[info] Using config /tmp/sc-ySbkbNvt3/fmt/mull.yml
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Filter mutants (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Running mutants (threads: 8)
       [################################] 96/96. Finished in 312ms
[info] Survived mutants (92/96):
/tmp/sc-ySbkbNvt3/fmt/include/fmt/core.h:2237:14: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
```

(continues on next page)

(continued from previous page)

```
  return len + !len;
              ^
/tmp/sc-ySbkbNvt3/fmt/include/fmt/core.h:2267:24: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
    value = value * 10 + unsigned(*p - '0');
                       ^
/tmp/sc-ySbkbNvt3/fmt/include/fmt/core.h:2277:31: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
                 prev * 10ull + unsigned(p[-1] - '0') <= max
                              ^
<truncated>
/tmp/sc-ySbkbNvt3/fmt/include/fmt/format.h:2335:40: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
  if (negative) abs_value = ~abs_value + 1;
                                         ^
/tmp/sc-ySbkbNvt3/fmt/include/fmt/format.h:2337:34: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
  auto size = (negative ? 1 : 0) + static_cast<size_t>(num_digits);
                                 ^
[info] Mutation score: 4%
[info] Total execution time: 368ms
```

We get fewer mutants, but the number can be reduced even further.

5.3.5 Step 5. Filter out unreachable mutants

This step left as an exercise for the reader: *Keeping mutants under control*.

Hint: use Code Coverage filter.

5.4 Keeping mutants under control

This tutorial shows how to control the amount of mutants.

—

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.

5.4.1 OpenSSL Example

The *OpenSSL* tutorial makes a great example of when we want to decrease the amount of mutants.

To recap, recreate the same setup.

1. Checkout OpenSSL:

```
git clone https://github.com/openssl/openssl.git \
  --branch openssl-3.0.1 \
  --depth 1
```

2. Create Mull config file `openssl/mull.yml`:

```
mutators:
  - cxx_add_to_sub
```

3. Build OpenSSL:

```
cd openssl
export CC=clang-12
./config -O0 -fexperimental-new-pass-manager \
  -fpass-plugin=/usr/lib/mull-ir-frontend-12 \
  -g -grecord-command-line
make build_generated -j
make ./test/bio_enc_test -j
```

4. Run Mull:

```
$ mull-runner-12 ./test/bio_enc_test
[info] Using config /tmp/sc-g6cD7gfN4/openssl/mull.yml
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 638ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 281ms
[info] Running mutants (threads: 8)
       [################################] 1606/1606. Finished in 147786ms
[info] Survived mutants (1588/1606):
/tmp/sc-g6cD7gfN4/openssl/apps/lib/opt.c:1126:15: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
        i = 2 + (int)strlen(o->name);
              ^
/tmp/sc-g6cD7gfN4/openssl/apps/lib/opt.c:1128:20: warning: Survived: Replaced + with -␣
↪[cxx_add_to_sub]
            i += 1 + strlen(valtype2param(o));
                   ^
/tmp/sc-g6cD7gfN4/openssl/crypto/aria/aria.c:546:20: warning: Survived: Replaced + with -
↪ [cxx_add_to_sub]
    int Nr = (bits + 256) / 32;
                   ^
/tmp/sc-g6cD7gfN4/openssl/crypto/asn1/a_bitstr.c:62:13: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    ret = 1 + len;
            ^
<truncated>
/tmp/sc-g6cD7gfN4/openssl/test/testutil/format_output.c:282:47: warning: Survived:␣
↪Replaced + with - [cxx_add_to_sub]
    l1 = bn1 == NULL ? 0 : (BN_num_bytes(bn1) + (BN_is_negative(bn1) ? 1 : 0));
                                               ^
/tmp/sc-g6cD7gfN4/openssl/test/testutil/format_output.c:283:47: warning: Survived:␣
↪Replaced + with - [cxx_add_to_sub]
    l2 = bn2 == NULL ? 0 : (BN_num_bytes(bn2) + (BN_is_negative(bn2) ? 1 : 0));
                                               ^
/tmp/sc-g6cD7gfN4/openssl/test/testutil/format_output.c:301:32: warning: Survived:␣
↪Replaced + with - [cxx_add_to_sub]
    len = ((l1 > l2 ? l1 : l2) + bytes - 1) / bytes * bytes;
```

(continues on next page)

(continued from previous page)

```
                                ^
/tmp/sc-g6cD7gfN4/openssl/test/testutil/random.c:24:54: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    test_random_state[pos] += test_random_state[(pos + 28) % 31];
                                                       ^
[info] Mutation score: 1%
[info] Total execution time: 149344ms
```

In the end, you should see about ~1.5k survived mutants.

There are at least two kinds of "problematic" mutants there:

- not interesting: e.g., we probably don't care about mutants under `testutil`
- unreachable: the test suite cannot detect them

Let's try to fix these issues one by one.

5.4.2 File Path Filters

First, let's tell Mull to not mutate and not to run Mutants under `testutil`.

We can extend the same `mull.yml` file we used to configure Mull at the very beginning.

Mull comes with two path-based filters: `excludePaths` and `includePaths`. You can use these to either exclude or include mutations based on their file-system location. To ignore any mutants under `testutil` edit `mull.yml` as follows:

```
mutators:
  - cxx_add_to_sub
excludePaths:
  - .*testutil.*
```

Now, rerun Mull:

```
$ mull-runner-12 ./test/bio_enc_test
[info] Using config /tmp/sc-g6cD7gfN4/openssl/mull.yml
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 282ms
[info] Filter mutants (threads: 1)
       [################################] 1/1. Finished in 2ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 283ms
[info] Running mutants (threads: 8)
       [################################] 1585/1585. Finished in 149522ms
[info] Survived mutants (1568/1585):
<truncated>
[info] Mutation score: 1%
[info] Total execution time: 150815ms
```

Note: Some config options understood by both `mull-ir-frontend` and `mull-runner`. In this case, we don't need to recompile the program under test - `mull-runner` picks up the config changes and filters out not needed mutants.

Though, `./test/bio_enc_test` still contains the mutants from `testutil`, they are just ignored.

Total 1585 mutants vs 1606 previously. Slightly better, but still not great.

We need something heavier than that!

5.4.3 Code Coverage Filter

Mull understands code coverage, but for that to work we should recompile OpenSSL to include the instrumentation information:

```
make clean
./config -O0 -fexperimental-new-pass-manager \
  -fpass-plugin=/usr/lib/mull-ir-frontend-12 \
  -g -grecord-command-line \
  -fprofile-instr-generate -fcoverage-mapping
make build_generated -j
make ./test/bio_enc_test -j
```

Note: This time, `mull-ir-frontend` picks up `excludePaths` from `mull.yml` and `./test/bio_enc_test` no longer contains mutations from `testutil`.

Rerun Mull:

```
$ mull-runner-12 ./test/bio_enc_test
[info] Using config /tmp/sc-g6cD7gfN4/openssl/mull.yml
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 1281ms
[info] Extracting coverage information (threads: 1)
       [################################] 1/1. Finished in 361ms
[info] Filter mutants (threads: 1)
       [################################] 1/1. Finished in 36ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 326ms
[info] Running mutants (threads: 8)
       [################################] 34/34. Finished in 7805ms
[info] Survived mutants (18/34):
/tmp/sc-g6cD7gfN4/openssl/crypto/conf/conf_mod.c:556:22: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    size = strlen(t) + strlen(sep) + strlen(OPENSSL_CONF) + 1;
                     ^
/tmp/sc-g6cD7gfN4/openssl/crypto/conf/conf_mod.c:556:36: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    size = strlen(t) + strlen(sep) + strlen(OPENSSL_CONF) + 1;
                                   ^
<truncated>
/tmp/sc-g6cD7gfN4/openssl/providers/implementations/rands/drbg_ctr.c:427:37: warning:␣
↪Survived: Replaced + with - [cxx_add_to_sub]
        ctr32 = GETU32(ctr->V + 12) + blocks;
                                    ^
/tmp/sc-g6cD7gfN4/openssl/providers/implementations/rands/drbg_ctr.c:555:28: warning:␣
↪Survived: Replaced + with - [cxx_add_to_sub]
    drbg->seedlen = keylen + 16;
                           ^
```

(continues on next page)

(continued from previous page)

```
/tmp/sc-g6cD7gfN4/openssl/providers/implementations/rands/seed_src.c:191:44: warning:␣
↪Survived: Replaced + with - [cxx_add_to_sub]
    bytes_needed = entropy >= 0 ? (entropy + 7) / 8 : 0;
                                           ^
[info] Mutation score: 47%
[info] Total execution time: 12449ms
```

A few things worth mentioning here:

- there is a new running phase `Extracting coverage information`: Mull handles code coverage info automatically
- we've got 34 mutants instead of ~1.5k
- total execution time dropped from ~150 seconds to only 12 seconds

With this improvement in place there are two ways forward:

1. Extend the test suite to ensure there are no survived mutants
2. Add more *mutators* and go to the step 1 above.

5.5 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.

5.5.1 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;
  }
  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");
```

(continues on next page)

(continued from previous page)

```
      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
```

Usage of Mull in this case is very similar to a "typical" use-case (see *Hello World tutorial*).

1. Create config file `mull.yml`:

```
mutators:
 - cxx_add_to_sub
 - cxx_mul_to_div
```

2. Generate mutated executable

```
$ clang-12 -fexperimental-new-pass-manager \
  -fpass-plugin=/usr/local/lib/mull-ir-frontend-12 \
  -g -grecord-command-line \
  main.c -o test.exe
```

3. Run analysis using *mull-runner*:

```
$ mull-runner-12 ./test.exe -ide-reporter-show-killed \
  -test-program=python3 -- test.py ./test.exe
[info] Using config /tmp/sc-kGN35Gr1f/mull.yml
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 347ms
[info] Filter mutants (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 76ms
[info] Running mutants (threads: 2)
       [################################] 2/2. Finished in 81ms
[info] Killed mutants (2/2):
/tmp/sc-kGN35Gr1f/main.c:5:12: warning: Killed: Replaced + with - [cxx_add_to_sub]
  return a + b;
           ^
/tmp/sc-kGN35Gr1f/main.c:9:12: 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: 509ms
```

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

5.6 Working with SQLite report

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
- `PatchesReporter`: saves each mutant as a `.patch` that can be applied on the original code

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.

5.6.1 Preparing the database

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-runner-12 --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 line
```

5.6.2 Database Schema

Now you can examine contents of the database:

```
sqlite> .tables
information  mutant

sqlite> .schema information
CREATE TABLE information (
  key TEXT,
  value TEXT
);
sqlite> .schema mutant
CREATE TABLE mutant (
  mutator TEXT,
  filename TEXT,
  directory TEXT,
  line_number INT,
  column_number INT,
  status INT,
  duration INT,
  stdout TEXT,
  stderr TEXT
);
```

The database contains two tables: `mutant` and `information`.

The `information` table stores a number of key/value pairs with certain facts about Mull:

```
sqlite> select * from information;
  key = LLVM Version
value = 12.0.1
```

(continues on next page)

(continued from previous page)

```
  key = Build Date
value = 08 Mar 2022

  key = Commit
value = 8f01ac4d

  key = Mull Version
value = 0.16.0

  key = URL
value = https://github.com/mull-project/mull
```

And the `mutant` table stores the name of the mutation operator, the location of the mutant, and information about the execution of each mutant: duration, status (passed, failed, etc) and the text from standard out and err streams.

```
sqlite> select * from mutant limit 1;
    mutant_id = cxx_add_to_sub:/tmp/sc-76UJhQXB4/fmt/include/fmt/core.h:822:23
      mutator = cxx_add_to_sub
     filename = /tmp/sc-76UJhQXB4/fmt/include/fmt/core.h
    directory =
  line_number = 822
column_number = 23
       status = 1
     duration = 14
       stdout = [==========] Running 55 tests from 19 test suites.
<truncated>
       stderr =
```

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

Basic exploration

Let's check how many mutants:

```
sqlite>  select count(*) from mutant;
count(*) = 163
```

Let's see some stats on the execution time:

```
sqlite> select avg(duration), max(duration) from mutant;
avg(duration) = 10.5276073619632
```

(continues on next page)

(continued from previous page)

```
max(duration) = 104
```

Let's see what's wrong with that slow run:

```
sqlite> select mutant_id, status, duration from mutant order by duration desc limit 1;
mutant_id = cxx_add_to_sub:/tmp/sc-76UJhQXB4/fmt/include/fmt/format.h:684:23
   status = 3
 duration = 104
```

The mutant status is 3, which is a timeout according to the table above.

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 a `mutant` is anything but `Passed`.

Counting all the mutants is rather trivial but a bit lengthy, so let's create an SQL view:

```
sqlite> create view killed_mutants as select * from mutant where status <> 2;
sqlite> select count(*) as killed from killed_mutants;
killed = 4
```

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 mutant) * 100) as score;
score = 2.0
```

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;
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

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

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-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

--ld-search-path directory Library search path

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

--debug-coverage Print coverage ranges

CHAPTER

NINE

CONFIGURING MULL

Mull's IR frontend is configured via a text file in the `yaml` format.

By default, Mull is looking for `mull.yml` file in the current directory. If it cannot find it, then it tries the parent directory and does so recursively until it finds the config file or reaches the root of the file system.

Alternatively, you can set `MULL_CONFIG` to point to the config file.

Here is an example config file:

```
mutators:
 - cxx_add_to_sub
 - cxx_logical
excludePaths: # support regex
 - gtest
 - gmock
timeout: # milliseconds
 - 10000 # 10 seconds
quiet: false # enables additional logging
```

CHAPTER

TEN

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).

10.1 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.

10.2 Mutations search

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

The **"IR 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 IR by the default search algorithm.

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

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

10.3 Supported mutation operators

See Supported Mutation Operators.

10.4 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.

10.5 Platform support

Mull has a great support of macOS and various Linux systems across all modern versions of LLVM from 9.0 to 13.0. All the new versions of LLVM are supported as soon as they released.

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

10.6 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)

10.7 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.

10.8 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.

10.9 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.

10.10 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},}
```

10.11 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

ELEVEN

HACKING ON MULL

11.1 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.

11.2 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
```

11.3 Local Development Setup

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

11.3.1 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.

11.3.2 LLVM

You need LLVM to build and debug Mull. You can use any LLVM version between 9.0 and 13.0.

As of the version 0.14.0, Mull can be compiled against LLVM/Clang available through your package manager (e.g. apt or homebrew).

11.3.3 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 -DCMAKE_PREFIX_PATH=<cmake search paths> ..
make mull-runner-12
make mull-tests
```

The `cmake search paths` should point to the LLVM/Clang CMake config folders. Some examples:

- llvm@12 installed via homebrew on macOS: `"/usr/local/opt/llvm@12/lib/cmake/llvm/;/usr/local/opt/llvm@12/lib/cmake/clang/"`
- llvm-12 installed via apt on Ubuntu: `"/usr/lib/llvm-13/cmake/;/usr/lib/cmake/clang-13/"`

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 -DCMAKE_PREFIX_PATH=<cmake search paths> -DCMAKE_CXX_FLAGS=-D_GLIBCXX_USE_CXX11_
↪ABI=0 ..
```