# Mull

***Release 0.8.0***

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

**Jan 02, 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.

Doing research? There is a special chapter for you: For Researchers.

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 Slack.

# CHAPTER TWO

# INTRODUCTION TO MUTATION TESTING

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

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

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

# CHAPTER THREE

# INSTALLATION

Mull comes with a number of precompiled binaries for macOS and Ubuntu.

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

## 3.1 Install on Ubuntu

```
wget https://api.bintray.com/users/bintray/keys/gpg/public.key
sudo apt-key add public.key
echo "deb https://dl.bintray.com/mull-project/ubuntu-18 stable main" | sudo tee -a /
↪etc/apt/sources.list
sudo apt-get update
sudo apt-get install mull
```

Check if everything works:

```
$ mull-cxx --version
Mull: LLVM-based mutation testing
https://github.com/mull-project/mull
Version: 0.7.0
Commit: 1638698
Date: 28 Mar 2020
LLVM: 9.0.0
```

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

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

## 3.2 Install on macOS

Get the latest version here Bintray.

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: 0.7.0
Commit: 1638698
Date: 28 Mar 2020
LLVM: 9.0.0
```

You can also get the latest “nightly” build from here.

# CHAPTER FOUR

# TUTORIALS

## 4.1 Hello World Example

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

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

---

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

```
$ mull-cxx -version
Mull: LLVM-based mutation testing
https://github.com/mull-project/mull
Version: 0.7.0
Commit: 1638698
Date: 28 Mar 2020
LLVM: 9.0.0
```

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

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

and compile it:

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

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

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

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

We specify `CustomTest`:

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

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

Now the output is different:

```
[info] Extracting bitcode from executable (threads: 1)
[warning] No bitcode: x86_64
       [################################] 1/1. Finished in 1ms
[info] Loading dynamic libraries (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Searching tests (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] No mutants found. Mutation score: infinitely high
```

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

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

Let's try again:

```
$ clang -fembed-bitcode -g main.cpp -o hello-world
$ mull-cxx -test-framework CustomTest hello-world
[info] Extracting bitcode from executable (threads: 1)
       [################################] 1/1. Finished in 3ms
[info] Loading bitcode files (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Compiling instrumented code (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Loading dynamic libraries (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Searching tests (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Preparing original test run (threads: 1)
       [################################] 1/1. Finished in 1ms
[info] Running original tests (threads: 1)
       [################################] 1/1. Finished in 12ms
[info] No mutants found. Mutation score: infinitely high
```

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

Let's add some code:

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

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

(continues on next page)

(continued from previous page)

```
    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 input but in this tutorial we want to be more verbose.

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

(continues on next page)

(continued from previous page)

```
       [################################] 1/1. Finished in 10ms
[info] Running mutants (threads: 2)
       [################################] 2/2. Finished in 12ms
[info] Killed mutants (1/2):
/tmp/sc-b3yQyijWP/main.cpp:2:11: warning: Killed: Replaced >= with < [cxx_ge_to_lt]
  if (age >= 21) {
          ^
[info] Survived mutants (1/2):
/tmp/sc-b3yQyijWP/main.cpp:2:11: warning: Survived: Replaced >= with > [cxx_ge_to_gt]
  if (age >= 21) {
          ^
[info] Mutation score: 50%
```

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

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

## 4.1.1 Complete example

The code:

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

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

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

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

  /// success
  return 0;
}
```

```
$ clang -fembed-bitcode -g main.cpp -o hello-world
$ mull-cxx -test-framework=CustomTest -ide-reporter-show-killed hello-world
[info] Extracting bitcode from executable (threads: 1)
       [################################] 1/1. Finished in 4ms
[info] Loading bitcode files (threads: 1)
       [################################] 1/1. Finished in 12ms
[info] Compiling instrumented code (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Loading dynamic libraries (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Searching tests (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Preparing original test run (threads: 1)
       [################################] 1/1. Finished in 1ms
[info] Running original tests (threads: 1)
       [################################] 1/1. Finished in 10ms
[info] Applying function filter: no debug info (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Applying function filter: file path (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Instruction selection (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Searching mutants across functions (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Applying filter: no debug info (threads: 2)
       [################################] 2/2. Finished in 0ms
[info] Applying filter: file path (threads: 2)
       [################################] 2/2. Finished in 0ms
[info] Prepare mutations (threads: 1)
       [################################] 1/1. Finished in 1ms
[info] Cloning functions for mutation (threads: 1)
       [################################] 1/1. Finished in 11ms
[info] Removing original functions (threads: 1)
       [################################] 1/1. Finished in 12ms
[info] Redirect mutated functions (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Applying mutations (threads: 1)
       [################################] 2/2. Finished in 0ms
[info] Compiling original code (threads: 1)
       [################################] 1/1. Finished in 12ms
[info] Running mutants (threads: 2)
       [################################] 2/2. Finished in 10ms
[info] Killed mutants (2/2):
/tmp/sc-b3yQyijWP/main.cpp:2:11: warning: Killed: Replaced >= with > [cxx_ge_to_gt]
  if (age >= 21) {
          ^
/tmp/sc-b3yQyijWP/main.cpp:2:11: warning: Killed: Replaced >= with < [cxx_ge_to_lt]
  if (age >= 21) {
          ^
[info] All mutations have been killed
[info] Mutation score: 100%
```

### 4.1.2 Summary

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

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

## 4.2 fmtlib tutorial

This tutorial will show you how to run Mull against fmtlib.

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
```

Run Mull against the `core-test`:

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

If everything works, you will see a number of confusing mutations within the report:

```
/opt/llvm/9.0.0/bin/../include/c++/v1/__tree:2114:5: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    ++size();
    ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format.h:1738:31: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    if (specs.width != 0) --specs.width;
                          ^
```

This is because of `Junk Mutations`.

### 4.2.1 Junk Mutations

Not every mutation found at Bitcode level can be represented at the source level. Mull can filter them out by looking at the source code, but for that you need to 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/5.0.0/lib/clang/5.0.0/include
 /usr/include
 /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/5.0.0/include/c++/v1
 /usr/local/include
 /opt/llvm/5.0.0/lib/clang/5.0.0/include
 /usr/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/5.0.0/include/c++/v1
/usr/local/include
/opt/llvm/5.0.0/lib/clang/5.0.0/include
/usr/include
```

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

```
mull-cxx -test-framework=GoogleTest \
  -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" \
    ./bin/core-test
```

You should see similar output:

```
[info] Extracting bitcode from executable (threads: 1)
       [################################] 1/1. Finished in 154ms
[info] Loading bitcode files (threads: 4)
       [################################] 4/4. Finished in 305ms
[info] Compiling instrumented code (threads: 4)
       [################################] 4/4. Finished in 5628ms
[info] Loading dynamic libraries (threads: 1)
       [################################] 1/1. Finished in 0ms
```

(continues on next page)

(continued from previous page)

```
[info] Searching tests (threads: 1)
       [################################] 1/1. Finished in 2ms
[info] Preparing original test run (threads: 1)
       [################################] 1/1. Finished in 96ms
[info] Running original tests (threads: 8)
       [################################] 36/36. Finished in 92ms
[info] Applying function filter: no debug info (threads: 8)
       [################################] 3624/3624. Finished in 13ms
[info] Applying function filter: file path (threads: 8)
       [################################] 3624/3624. Finished in 11ms
[info] Instruction selection (threads: 8)
       [################################] 3624/3624. Finished in 27ms
[info] Searching mutants across functions (threads: 8)
       [################################] 3624/3624. Finished in 11ms
[info] Applying filter: no debug info (threads: 8)
       [################################] 102/102. Finished in 0ms
[info] Applying filter: file path (threads: 8)
       [################################] 102/102. Finished in 1ms
[info] Applying filter: junk (threads: 8)
       [################################] 102/102. Finished in 3397ms
[info] Prepare mutations (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Cloning functions for mutation (threads: 4)
       [################################] 4/4. Finished in 45ms
[info] Removing original functions (threads: 4)
       [################################] 4/4. Finished in 22ms
[info] Redirect mutated functions (threads: 4)
       [################################] 4/4. Finished in 13ms
[info] Applying mutations (threads: 1)
       [################################] 41/41. Finished in 13ms
[info] Compiling original code (threads: 4)
       [################################] 4/4. Finished in 3901ms
[info] Running mutants (threads: 8)
       [################################] 41/41. Finished in 578ms
[info] Survived mutants (22/41):
/tmp/sc-UBe3GUa96/fmt/test/gmock-gtest-all.cc:9758:53: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    const int actual_to_skip = stack_frames_to_skip + 1;
                                                    ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format.h:1466:42: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    if (negative) abs_value = ~abs_value + 1;
                                         ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:843:53: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        (static_cast<uint64_t>(integral) << -one.e) + fractional;
                                                    ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:854:31: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        static_cast<char>('0' + static_cast<char>(fractional >> -one.e));
                              ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:1096:33: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        min_exp - (normalized.e + fp::significand_size), cached_exp10);
                                ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:843:53: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
```

(continues on next page)

(continued from previous page)

```
        (static_cast<uint64_t>(integral) << -one.e) + fractional;
                                                    ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:844:53: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    result = handler.on_digit(static_cast<char>('0' + digit),
                                                    ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:854:31: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        static_cast<char>('0' + static_cast<char>(fractional >> -one.e));
                              ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format.h:1678:30: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    auto&& it = reserve(size + padding * fill_size);
                             ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:415:34: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    fp upper = normalize<0>(fp(f + half_ulp, e));
                                 ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:466:69: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
      ((min_exponent + fp::significand_size - 1) * one_over_log2_10 +
                                                                    ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:946:27: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    uint64_t down = (diff + 1) * unit;  // wp_Wdown
                          ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:948:20: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        (remainder + divisor < down ||
                   ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:949:39: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
         down - remainder > remainder + divisor - down)) {
                                      ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format.h:1184:31: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    int full_exp = num_digits + exp - 1;
                              ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:446:67: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
  return (static_cast<uint64_t>(product) & (1ULL << 63)) != 0 ? f + 1 : f;
                                                                  ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:932:20: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        (remainder + divisor < d || d - remainder >= remainder + divisor - d)) {
                   ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format-inl.h:932:64: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
        (remainder + divisor < d || d - remainder >= remainder + divisor - d)) {
                                                               ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format.h:886:54: warning: Survived: Replaced + with␣
↪- [cxx_add_to_sub]
    *--buffer = static_cast<Char>(data::digits[index + 1]);
                                                     ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format.h:1466:42: warning: Survived: Replaced +␣
↪with - [cxx_add_to_sub]
    if (negative) abs_value = ~abs_value + 1;
```

(continues on next page)

(continued from previous page)

```
                                               ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format.h:886:54: warning: Survived: Replaced + with␣
↪- [cxx_add_to_sub]
    *--buffer = static_cast<Char>(data::digits[index + 1]);
                                                     ^
/tmp/sc-UBe3GUa96/fmt/include/fmt/format.h:892:39: warning: Survived: Replaced + with␣
↪- [cxx_add_to_sub]
    *--buffer = static_cast<Char>('0' + value);
                                      ^
[info] Mutation score: 46%
```

Now the mutants are valid and point to the right places.

## 4.3 OpenSSL tutorial

This tutorial will show you how to run Mull against OpenSSL. This tutorial is similar to the fmtlib tutorial, but there are two key differences:

- fmtlib uses CMake-based build system, while OpenSSL' build system is very custom and constitutes of a number of shell-scripts
- fmtlib is written in C++, while OpenSSL has some (optional) assembly code
- fmtlib's build system gives us a nice, ready to use compilation database, while for OpenSSL we need to mimic it manually

Let's get started!

### 4.3.1 Build OpenSSL

Get sources, configure the build system and build everything:

```
git clone https://github.com/openssl/openssl.git
cd openssl
export CC=clang
./config -no-asm -no-shared -no-module -no-des -no-threads -g -O0 -fembed-bitcode
make all
```

Some parts of OpenSSL are written in assembly, but Mull requires LLVM Bitcode to run the program under JIT environment. To avoid assembly we ask OpenSSL to no use assembly, but instead fall back to identical C implementation.

If we omit `-no-shared` and `-no-module` flags, then OpenSSL will build `libcrypto.dylib` (or `.so` on Linux) and link all the test executables against the dynamic library. In this case, the test executable will only contain LLVM Bitcode for the tests, but not for the OpenSSL core: the rest of bitcode is in `libcrypto.dylib`, which can still be accessed by Mull, but this feature is not implemented yet. By disabling shared library we force the build system to build self-contained executables.

JITting code that uses `pthreads` brings some issues that we were not able to debug and fix yet, so we disable threads as well.

The rest of build flags ask OpenSSL to emit debug info, to not run optimizations, and finally to embed LLVM Bitcode into the binary.

*Note: without -no-des the build fails for some awkward reason which we were to lazy to debug and fix.*

### 4.3.2 Examine OpenSSL

Let's examine `bio_enc_test` test suite:

```
mull-cxx -test-framework=CustomTest -mutators=cxx_lshift_assign_to_rshift_assign test/
↪bio_enc_test
```

If you run it on Linux you should see the following warnings:

```
[warning] Could not find dynamic library: libdl.so.2
[warning] Could not find dynamic library: libc.so.6
```

In order to run tests (original or mutated) Mull needs to feed the whole program and all its dependencies into the JIT engine. This includes dynamic libraries. Mull extracts them from the executable. On macOS, in most of the cases, the dynamic libraries have the full path. However, on Linux there are only names of the libraries, while the full paths normally determined by the dynamic linker `ldd`. Mull does not assume any of the paths, so this is responsibility of the user to provide the paths.

macOS example:

```
> otool -L test/bio_enc_test
test/bio_enc_test:
    /usr/lib/libSystem.B.dylib
```

Linux example:

```
> ldd test/bio_enc_test
linux-vdso.so.1 (0x00007fffb01e0000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f4df0be8000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4df05d8000)
/lib64/ld-linux-x86-64.so.2 (0x00007f4df0dec000)
```

In this example, On Linux, required libraries located under `/lib/x86_64-linux-gnu`, so we instruct Mull about the paths via `-ld-search-path` CLI option:

```
mull-cxx -test-framework=CustomTest -mutators=cxx_lshift_assign_to_rshift_assign -ld-
↪search-path=/lib/x86_64-linux-gnu test/bio_enc_test
```

### 4.3.3 Junk Mutations

Now everything works great, but there is one issue: we asked Mull to only apply `cxx_lshift_assign_to_rshift_assign` mutations, which replaces `<<=` with `>>=` (see the `-mutators=cxx_lshift_assign_to_rshift_assign` option), but there are some weird results:

```
> mull-cxx -test-framework=CustomTest -mutators=cxx_lshift_assign_to_rshift_assign -
↪ld-search-path=/lib/x86_64-linux-gnu test/bio_enc_test
[info] Extracting bitcode from executable (threads: 1)
       [################################] 1/1. Finished in 268ms
...
skipped
...
[info] Survived mutants (91/128):
/home/vagrant/openssl/test/testutil/driver.c:335:30: warning: Survived: Replaced <<=␣
↪with >>= [cxx_lshift_assign_to_rshift_assign]
            if (all_tests[i].subtest && single_iter == -1) {
                             ^
```

(continues on next page)

(continued from previous page)

```
/home/vagrant/openssl/test/testutil/driver.c:369:34: warning: Survived: Replaced <<=␣
↪with >>= [cxx_lshift_assign_to_rshift_assign]
                if (all_tests[i].subtest) {
                                 ^
/home/vagrant/openssl/test/testutil/driver.c:74:34: warning: Survived: Replaced <<=␣
↪with >>= [cxx_lshift_assign_to_rshift_assign]
    all_tests[num_tests].subtest = subtest;
                                 ^
...
skipped
...
```

This is because not all of the mutations available on the bitcode level can be represented on the source code level. These are called Junk Mutations. In order to solve this problem and get reasonable results we need to instruct

This is because not every mutation found at Bitcode level can be represented at the source level. Mull can filter them out by looking at the source code, but for that you need to provide compilation database, or compilation flags, or both.

In case of the custom build system it is not trivial to get the compilation database, so we have to hand-craft the compilation flags ourselves.

**Please, note:** Clang adds implicit header search paths, which must be added explicitly via `-compilation-flags`. You can get them using the following command:

```
> clang -x c -c /dev/null -v
... skipped
#include <...> search starts here:
 /usr/local/include
 /opt/llvm/5.0.0/lib/clang/5.0.0/include
 /usr/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:

```
/usr/local/include
/usr/lib/llvm-6.0/lib/clang/6.0.0/include
/usr/include/x86_64-linux-gnu
/usr/include
```

The final command to run Mull looks like this:

```
> mull-cxx -test-framework=CustomTest \
 -mutators=cxx_lshift_assign_to_rshift_assign \
 -ld-search-path=/lib/x86_64-linux-gnu \
 -compilation-flags="\
   -D_REENTRANT -DMODULESDIR=\"/usr/local/lib/ossl-modules\"
   -I . -I crypto/modes -I crypto/include -I include -I apps/include \
   -I providers/implementations/include -I providers/common/include \
   -isystem /usr/local/include \
   -isystem /usr/lib/llvm-6.0/lib/clang/6.0.0/include \
   -isystem /usr/include \
   -isystem /usr/include/x86_64-linux-gnu" \
 test/bio_enc_test
```

If everything is correct, then you will see very similar output:

```
[info] Extracting bitcode from executable (threads: 1)
       [################################] 1/1. Finished in 277ms
[info] Loading bitcode files (threads: 2)
       [################################] 690/690. Finished in 734ms
[info] Compiling instrumented code (threads: 2)
       [################################] 690/690. Finished in 11ms
[info] Loading dynamic libraries (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Searching tests (threads: 1)
       [################################] 1/1. Finished in 4ms
[info] Preparing original test run (threads: 1)
       [################################] 1/1. Finished in 179ms
[info] Running original tests (threads: 1)
       [################################] 1/1. Finished in 613ms
[info] Applying function filter: no debug info (threads: 2)
       [################################] 833/833. Finished in 11ms
[info] Applying function filter: file path (threads: 2)
       [################################] 833/833. Finished in 11ms
[info] Instruction selection (threads: 2)
       [################################] 833/833. Finished in 21ms
[info] Searching mutants across functions (threads: 2)
       [################################] 833/833. Finished in 11ms
[info] Applying filter: no debug info (threads: 2)
       [################################] 128/128. Finished in 11ms
[info] Applying filter: file path (threads: 2)
       [################################] 128/128. Finished in 10ms
[info] Applying filter: junk (threads: 2)
       [################################] 128/128. Finished in 659ms
[info] Prepare mutations (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Cloning functions for mutation (threads: 2)
       [################################] 690/690. Finished in 10ms
[info] Removing original functions (threads: 2)
       [################################] 690/690. Finished in 11ms
[info] Redirect mutated functions (threads: 2)
       [################################] 690/690. Finished in 0ms
[info] Applying mutations (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Compiling original code (threads: 2)
       [################################] 690/690. Finished in 23ms
[info] Running mutants (threads: 1)
       [################################] 1/1. Finished in 630ms
[info] Survived mutants (1/1):
/home/vagrant/openssl/crypto/sparse_array.c:96:25: warning: Survived: Replaced <<=␣
↪with >>= [cxx_lshift_assign_to_rshift_assign]
                    idx <<= OPENSSL_SA_BLOCK_BITS;
                        ^
[info] Mutation score: 0%
```

## 4.4 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

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.

### 4.4.1 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 |

### 4.4.2 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 -test-framework=GoogleTest \
  -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;
commit;
vacuum;
```

CHAPTER

# FIVE

# SUPPORTED MUTATION OPERATORS

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

continues on next page

Table 1 – continued from previous page

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

CHAPTER

SIX

# COMMAND LINE REFERENCE

**--workers number** How many threads to use

**--timeout number** Timeout per test run (milliseconds)

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

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

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

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

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

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

**--reporters reporter** Choose reporters:

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

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

**--debug** Enables Debug Mode: more logs are printed

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

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

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

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

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

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

**--ld-preload library** Load the given libraries before dynamic linking

**--ld-search-path directory** Library search path

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

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

**--sandbox sandbox** Choose sandbox approach:

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

**Timer** Fastest, Recommended

**--test-framework framework** Choose test framework:

**GoogleTest** Google Test Framework

**CustomTest** Custom Test Framework

**CppUTest** CppUTest Framework

**SimpleTest** Simple Test (For internal usage only)

**--mutators mutator** Choose mutators:

**Groups:**

**all** cxx_all, experimental

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

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

**cxx_arithmetic_assignment** cxx_add_assign_to_sub_assign, cxx_sub_assign_to_add_assign, cxx_mul_assign_to_div_assign, cxx_div_assign_to_mul_assign, cxx_rem_assign_to_div_assign

**cxx_assignment** cxx_bitwise_assignment, cxx_arithmetic_assignment, cxx_const_assignment

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

**cxx_bitwise_assignment** cxx_and_assign_to_or_assign, cxx_or_assign_to_and_assign, cxx_xor_assign_to_or_assign, cxx_lshift_assign_to_rshift_assign, cxx_rshift_assign_to_lshift_assign

**cxx_boundary** cxx_le_to_lt, cxx_lt_to_le, cxx_ge_to_gt, cxx_gt_to_ge

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

**cxx_const_assignment** cxx_assign_const, cxx_init_const

**cxx_decrement** cxx_pre_dec_to_pre_inc, cxx_post_dec_to_post_inc

**cxx_default** cxx_increment, cxx_arithmetic, cxx_comparison, cxx_boundary

**cxx_increment** cxx_pre_inc_to_pre_dec, cxx_post_inc_to_post_dec

**cxx_logical** cxx_logical_and_to_or, cxx_logical_or_to_and, cxx_remove_negation

**experimental** negate_mutator, remove_void_function_mutator, scalar_value_mutator, replace_call_mutator, cxx_logical

**Single mutators:**

**cxx_add_assign_to_sub_assign** Replaces += with -=

**cxx_add_to_sub** Replaces + with -

**cxx_and_assign_to_or_assign** Replaces &= with |=

**cxx_and_to_or** Replaces & with |

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

**cxx_bitwise_not_to_noop** Replaces ~x with x

**cxx_div_assign_to_mul_assign** Replaces /= with *=

**cxx_div_to_mul** Replaces / with *

**cxx_eq_to_ne** Replaces == with !=

**cxx_ge_to_gt** Replaces >= with >

**cxx_ge_to_lt** Replaces >= with <

**cxx_gt_to_ge** Replaces > with >=

**cxx_gt_to_le** Replaces > with <=

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

**cxx_le_to_gt** Replaces <= with >

**cxx_le_to_lt** Replaces <= with <

**cxx_logical_and_to_or** Replaces && with ||

**cxx_logical_or_to_and** Replaces || with &&

**cxx_lshift_assign_to_rshift_assign** Replaces <<= with >>=

**cxx_lshift_to_rshift** Replaces << with >>

**cxx_lt_to_ge** Replaces < with >=

**cxx_lt_to_le** Replaces < with <=

**cxx_minus_to_noop** Replaces -x with x

**cxx_mul_assign_to_div_assign** Replaces *= with /=

**cxx_mul_to_div** Replaces * with /

**cxx_ne_to_eq** Replaces != with ==

**cxx_or_assign_to_and_assign** Replaces |= with &=

**cxx_or_to_and** Replaces | with &

**cxx_post_dec_to_post_inc** Replaces x– with x++

**cxx_post_inc_to_post_dec** Replaces x++ with x–

**cxx_pre_dec_to_pre_inc** Replaces –x with ++x

**cxx_pre_inc_to_pre_dec** Replaces ++x with –x

**cxx_rem_assign_to_div_assign** Replaces %= with /=

**cxx_rem_to_div** Replaces % with /

**cxx_remove_negation** Replaces !a with a

**cxx_rshift_assign_to_lshift_assign** Replaces >>= with <<=

**cxx_rshift_to_lshift** Replaces << with >>

**cxx_sub_assign_to_add_assign** Replaces -= with +=

**cxx_sub_to_add** Replaces - with +

**cxx_xor_assign_to_or_assign** Replaces ^= with |=

**cxx_xor_to_or** Replaces ^ with |

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

**remove_void_function_mutator** Removes calls to a function returning void

**replace_call_mutator** Replaces call to a function with 42

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

CHAPTER

SEVEN

# FOR RESEARCHERS

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

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

---

## 7.1 Design

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

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

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

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

## 7.2 Mutations search

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

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

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

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

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

## 7.3 Supported mutation operators

See Supported Mutation Operators.

## 7.4 Reporting

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

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

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

## 7.5 Platform support

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

Mull supports FreeBSD with minor known issues.

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

## 7.6 Test coverage

Mull has 3 layers of testing:

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

## 7.7 Current development

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

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

## 7.8 Advantages

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

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

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

## 7.9 Known issues

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

## 7.10 Paper

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

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

## 7.11 Additional information about Mull

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

CHAPTER

EIGHT

# HACKING ON MULL

## 8.1 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.

## 8.2 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
```

## 8.3 Local Development Setup

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

### 8.3.1 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.

### 8.3.2 LLVM

You need LLVM to build and debug Mull. You can use any LLVM version between 3.9 and 8.0.

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.**

### 8.3.3 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 ..
```