

# **LLVM-based mutation testing for C and C++**

**Alex Denisov, Virtual LLVM Dev Meeting 2020**

Hello everyone, thanks for coming to my talk :)

In this session I want to tell you about a LLVM-based tool for mutation testing. The tool is called Mull and available on Github. But before we get into the details, let me first address the question “What is mutation testing?”.

# What is Mutation Testing?

```
#include <assert.h>

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

int main() {
    // Associative: a + b = b + a
    assert(sum(5, 10) == sum(10, 5));

    // Commutative: a + (b + c) = (a + b) + c
    assert(sum(3, sum(4, 5)) == sum(sum(3, 4), 5));

    return 0;
}
```

The easiest way to answer this question is by showing an example. Let's say we need to write a function that calculates the sum of two numbers. Obviously, we need to test this function to make sure that the implementation is correct. In this case, we pick two mathematical properties. This test suite, however small, gives us green light. The tests are passing, code coverage is 100%, all good.

# What is Mutation Testing?

```
#include <assert.h>

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

int main() {
    // Associative: a + b = b + a
    assert(sum(5, 10) == sum(10, 5));

    // Commutative: a + (b + c) = (a + b) + c
    assert(sum(3, sum(4, 5)) == sum(sum(3, 4), 5));

    return 0;
}
```

```
#include <assert.h>

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

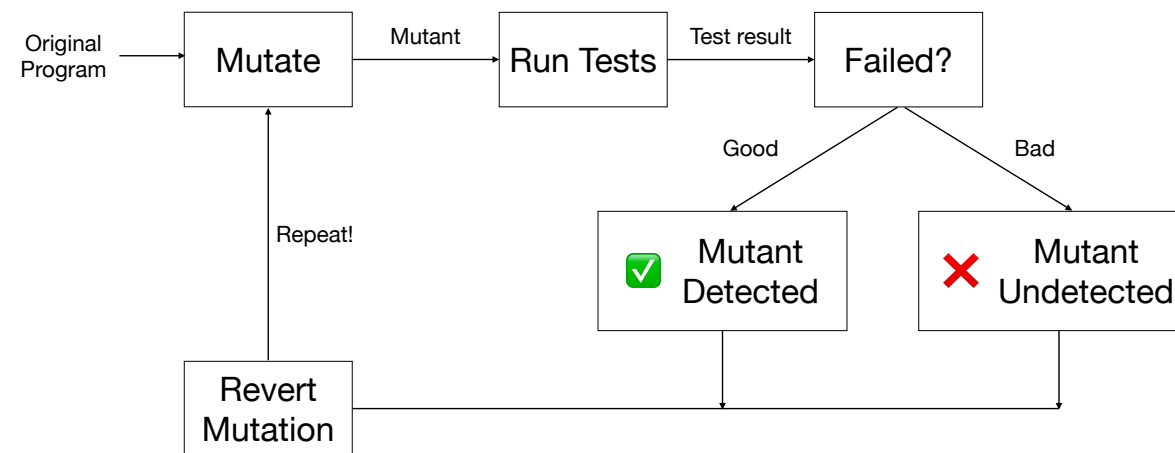
int main() {
    // Associative: a + b = b + a
    assert(sum(5, 10) == sum(10, 5));

    // Commutative: a + (b + c) = (a + b) + c
    assert(sum(3, sum(4, 5)) == sum(sum(3, 4), 5));

    return 0;
}
```

However, there is a problem. These mathematical properties also hold for multiplication. In this case, if we replace the plus with an asterisk, as on the right, then nothing special happens, the tests are still green. This change, or mutation, is not detected by the tests. Which means these tests are insufficient and should be extended. In a nutshell, this is the idea of mutation testing - introduce small semantic changes into the program and see if these changes are detected.

# What is Mutation Testing?



Schematically, it looks like this. You take the original program, add a mutation, run the tests against the mutated program and check the results. If the tests fail, then it is good because the change is detected. Otherwise, if the tests pass, then something is wrong, either with the tests or with the program itself.

This process can be repeated until you cannot come up with any new mutations.

# Mutation Operators

```
#include <assert.h>

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

int main() {
    // Associative: a + b = b + a
    assert(sum(5, 10) == sum(10, 5));

    // Commutative: a + (b + c) = (a + b) + c
    assert(sum(3, sum(4, 5)) == sum(sum(3, 4), 5));

    return 0;
}
```

## More mutations?

<pre>int sum(int a, int b) {     return a * b; }</pre>	<pre>int sum(int a, int b) {     return a; }</pre>
<pre>int sum(int a, int b) {     return a - b; }</pre>	<pre>int sum(int a, int b) {     return b; }</pre>
<pre>int sum(int a, int b) {     return a / b; }</pre>	<pre>int sum(int a, int b) {     return 0; }</pre>
<pre>int sum(int a, int b) {     return a % b; }</pre>	<pre>int sum(int a, int b) {     return 42; }</pre>

Speaking of which, there can be a lot. In mutation testing, each semantic change is controlled by so-called mutation operator.

Some examples are: replace addition with multiplication, replace addition with subtraction, replace binary operator with the first or the second operand, or replace an expression with a constant.

I can easily come up with 10 more distinct changes we can do to the code on the left. Obviously, doing this manually is a bad idea, so we should automate this process.

# Mull

## Practical mutation testing tool for C and C++

- Built with large projects in mind
- Transparent
- Deterministic\*
- Cross-platform (Linux, macOS, FreeBSD)
- Open Source  
<https://github.com/mull-project/mull>

So we did! I want to present you Mull, the tool for mutation testing which has C and C++ languages as its primary target.

Here is a list of items that I consider to be important and worth mentioning.

Mull is industry oriented, meaning that it can be used even with large projects, in such cases it can be applied incrementally and in case there is still too much work this work can be distributed across several machines.

Given the complexity of modern software we are trying to hide as much details from the user as possible, therefore simplifying the setup.

Mull's output is deterministic meaning that it will produce the same results over and over again. There are some exceptions to this statement, but I will cover it a bit later.

Also, Mull is cross-platform (if you don't count Windows as a platform). It also worked on FreeBSD at some point, but something went wrong and no one had time to fix it.

Ans last but not least - Mull is open source and available on Github.

## Why Mutation Testing?

- Evaluates quality of a test suite
- ???

You've got the idea, you've got the tool. But there is still a question of why one would want to use mutation testing. The consensus so far is that Mutation Testing is used to evaluate the quality of a test suite. This statement while correct is still a bit misleading and very limiting. Let me walk you through several findings I've seen in production.

# Example #1

## Code coverage report

```
Matrix4D::Matrix4D(double* arr) {  
1   int k = 0;  
10  for(int i = 0 ; i<4; i++) {  
40      for(int j = 0; j<4; j++) {  
16          r[i][j]=arr[k];  
16          k++;  
16      }  
4  }  
1 }  
  
Matrix4D::Matrix4D(const Matrix4D& other) {  
10  for(int i = 0 ; i<4; i++) {  
40      for(int j = 0; j<4; j++) {  
16          r[i][j]=other.r[i][j];  
16      }  
4  }  
1 }
```

```
void MATRIX4D_CONSTRUCTOR() {  
1   double matVals[] = {0.12, 3.45, 6.78, 9.01,  
                        2.34, 5.67, 8.90, 1.23,  
                        4.56, 7.89, 0.12, 3.45,  
                        6.78, 9.01, 2.34, 5.67};  
1   Matrix4D mat1(matVals);  
1   Matrix4D mat2(mat1);  
  
7   CPPUNIT_ASSERT(compareDouble(0.12, mat2.r[0][0]));  
7   CPPUNIT_ASSERT(compareDouble(3.45, mat2.r[1][0]));  
7   CPPUNIT_ASSERT(compareDouble(6.78, mat2.r[2][0]));  
7   CPPUNIT_ASSERT(compareDouble(9.01, mat2.r[3][0]));  
7   CPPUNIT_ASSERT(compareDouble(2.34, mat2.r[0][1]));  
7   CPPUNIT_ASSERT(compareDouble(5.67, mat2.r[1][1]));  
7   CPPUNIT_ASSERT(compareDouble(8.90, mat2.r[2][1]));  
7   CPPUNIT_ASSERT(compareDouble(1.23, mat2.r[3][1]));  
7   CPPUNIT_ASSERT(compareDouble(4.56, mat2.r[0][2]));  
7   CPPUNIT_ASSERT(compareDouble(7.89, mat2.r[1][2]));  
7   CPPUNIT_ASSERT(compareDouble(0.12, mat2.r[2][2]));  
7   CPPUNIT_ASSERT(compareDouble(3.45, mat2.r[3][2]));  
7   CPPUNIT_ASSERT(compareDouble(6.78, mat2.r[0][3]));  
7   CPPUNIT_ASSERT(compareDouble(9.01, mat2.r[1][3]));  
7   CPPUNIT_ASSERT(compareDouble(2.34, mat2.r[2][3]));  
7   CPPUNIT_ASSERT(compareDouble(5.67, mat2.r[3][3]));  
1 }
```

This example is coming from a proprietary operating system which is used in production in a couple of small aerospace missions. On the screenshot you can see two simple constructors of a 4-dimensional matrix on the left and the corresponding test case on the right. This view shows the code coverage report. The numbers on the left indicate that most of the lines were executed more than once. The test seems to be good.



# Example #1

## Mutation coverage report

```
1  #include "matlib.h"
2
3  Matrix4D::Matrix4D(double* arr) {
4      int k = 0;
5      for(int i = 0 ; i<4; i++) {
6          for(int j = 0; j<4; j++) {
7              r[i][j]=arr[k];
8              k++;
9          }
10     }
11 }
12
13 Matrix4D::Matrix4D(const Matrix4D& other) {
14     for(int i = 0 ; i<4; i++) {
15         for(int j = 0; j<4; j++) {
16             r[i][j]=other.r[i][j];
17         }
18     }
19 }
20 }
```

However, mutation testing shows very different picture. Here, the code marked with red and green colours highlight mutations. Green means that the mutation was detected, while the red means that the mutation survived. Most of the mutants in this code are not detected, which is bad. I must admit, when I first looked at this report I was disappointed. Given that the code has 100% coverage these mutations must have been detected, which means that Mull has a bug and shows the wrong result.

# Example #1

## The problem

```
bool compareDouble(bool left, bool right) {  
    return (left - THRESHOLD) < right && right < (left + THRESHOLD);  
}  
  
// compareDouble(0.12, 1)    -> true  
// compareDouble(0.12, 122) -> true  
// compareDouble(1000, 500) -> true  
// compareDouble(0, 0)      -> true  
// compareDouble(0, 100)    -> false  
// compareDouble(100, 0)    -> false
```

N.B. clang gives a warning, gcc does not

implicit conversion from 'double' to 'bool' changes value from 0.12 to true

But, when I started debugging the problem I found that the Mull is right and these mutations are indeed not detected. The problem, as it turned out, was in the `compareDouble` function.

For some reason, developers made a mistake and defined the input types as booleans, and not doubles. In this case, any non-zero value evaluated to true. And `compareDouble` returns false if and only if one of the arguments is zero.

# Example #1

## The solution

```
... @@ -5,7 +5,7 @@
5
6 #define THRESHOLD 0.001
7
8 -bool compareDouble(bool left, bool right) {
9     return (left - THRESHOLD) < right && right < (left + THR
10     ESHOLD);
11 }
12
13 ... @@ -23,20 +23,20 @@ void MyTestFixture::myTest() {
23     Matrix4D mat2(mat1);
24
25     CPPUNIT_ASSERT(compareDouble(0.12, mat2.r[0][0]));
26 - CPPUNIT_ASSERT(compareDouble(3.45, mat2.r[1][0]));
27 - CPPUNIT_ASSERT(compareDouble(6.78, mat2.r[2][0]));
28 - CPPUNIT_ASSERT(compareDouble(9.01, mat2.r[3][0]));
29 - CPPUNIT_ASSERT(compareDouble(2.34, mat2.r[0][1]));
30 - CPPUNIT_ASSERT(compareDouble(5.67, mat2.r[1][1]));
31 - CPPUNIT_ASSERT(compareDouble(8.90, mat2.r[2][1]));
32 - CPPUNIT_ASSERT(compareDouble(1.23, mat2.r[3][1]));
33 - CPPUNIT_ASSERT(compareDouble(4.56, mat2.r[0][2]));
34 - CPPUNIT_ASSERT(compareDouble(7.89, mat2.r[1][2]));
35 - CPPUNIT_ASSERT(compareDouble(0.12, mat2.r[2][2]));
36 - CPPUNIT_ASSERT(compareDouble(3.45, mat2.r[3][2]));
37 - CPPUNIT_ASSERT(compareDouble(6.78, mat2.r[0][3]));
38 - CPPUNIT_ASSERT(compareDouble(9.01, mat2.r[1][3]));
39 - CPPUNIT_ASSERT(compareDouble(2.34, mat2.r[2][3]));
40 - CPPUNIT_ASSERT(compareDouble(5.67, mat2.r[3][3]));
41 }
42
... @@ -5,7 +5,7 @@
5
6 #define THRESHOLD 0.001
7
8 +bool compareDouble(double left, double right) {
9     return (left - THRESHOLD) < right && right < (left + THR
10     ESHOLD);
11 }
12
13 ... @@ -23,20 +23,20 @@ void MyTestFixture::myTest() {
23     Matrix4D mat2(mat1);
24
25     CPPUNIT_ASSERT(compareDouble(0.12, mat2.r[0][0]));
26 + CPPUNIT_ASSERT(compareDouble(3.45, mat2.r[0][1]));
27 + CPPUNIT_ASSERT(compareDouble(6.78, mat2.r[0][2]));
28 + CPPUNIT_ASSERT(compareDouble(9.01, mat2.r[0][3]));
29 + CPPUNIT_ASSERT(compareDouble(2.34, mat2.r[1][0]));
30 + CPPUNIT_ASSERT(compareDouble(5.67, mat2.r[1][1]));
31 + CPPUNIT_ASSERT(compareDouble(8.90, mat2.r[1][2]));
32 + CPPUNIT_ASSERT(compareDouble(1.23, mat2.r[1][3]));
33 + CPPUNIT_ASSERT(compareDouble(4.56, mat2.r[2][0]));
34 + CPPUNIT_ASSERT(compareDouble(7.89, mat2.r[2][1]));
35 + CPPUNIT_ASSERT(compareDouble(0.12, mat2.r[2][2]));
36 + CPPUNIT_ASSERT(compareDouble(3.45, mat2.r[2][3]));
37 + CPPUNIT_ASSERT(compareDouble(6.78, mat2.r[3][0]));
38 + CPPUNIT_ASSERT(compareDouble(9.01, mat2.r[3][1]));
39 + CPPUNIT_ASSERT(compareDouble(2.34, mat2.r[3][2]));
40 + CPPUNIT_ASSERT(compareDouble(5.67, mat2.r[3][3]));
41 }
```

The fix for this problem was trivial - I just changed the types and... the test started to fail. The most interesting part is that almost all the tests that used compareDouble function were broken in one way or another.

## Example #2

```
int32_t File::read(char *buf, int32_t len) {
    if (filePtr->is_open()) {
        // actual reading
    }
    return -1;
}

int32_t File::getString(char *buf, int32_t max) {
    int32_t len = read(buf, max - 1);
    buf[len] = '\0';
    return len;
}

void ReadEmptyFileTest() {
    File f;
    f.open("an_empty_file");

    char buf[4] = { 0 };

    f.getString(buf, 4);

    CPPUNIT_ASSERT(buf[0], '\0');
    CPPUNIT_ASSERT(buf[1], '\0');
    CPPUNIT_ASSERT(buf[2], '\0');
    CPPUNIT_ASSERT(buf[3], '\0');
}
```

Here is the other example. In this case the class File represents an abstraction of a file system, that has different implementations when you run it Linux or macOS and an embedded device. The code on the left is just a snippet we are interested in. And on the right there is a simple test case.

The test checks that reading a string from an empty file doesn't change the buffer.

I guess you can see the problem already.

## Example #2

```
int32_t File::read(char *buf, int32_t len) {
    if (filePtr->is_open()) {
        // actual reading
    }
    return -1;
}

int32_t File::getString(char *buf, int32_t max) {
    int32_t len = read(buf, max - 1); // len = -1
    buf[len] = '\0';                  // buf[-1] = 0
    return len;
}

void ReadEmptyFileTest() {
    File f;
    // f.open("an_empty_file");

    char buf[4] = { 0 };

    f.getString(buf, 4);

    CPPUNIT_ASSERT(buf[0], '\0');
    CPPUNIT_ASSERT(buf[1], '\0');
    CPPUNIT_ASSERT(buf[2], '\0');
    CPPUNIT_ASSERT(buf[3], '\0');
}
```

Anyhow, what Mull did is basically removed the call to the open as indicated on the right. What happens then, is that the getString function calls the read function, the read returns -1, and getString successfully writes zero into the minus first element of the buffer. But the test is still passing. Sometimes.

## Example #3

```
int32_t bitstuff(int32_t number) {  
    number |= (1 << 7);  
    number |= (1 << 15);  
    number |= (1 << 23);  
    number |= (1 << 31);  
    return number;  
}
```

```
int32_t bitstuff(int32_t number) {  
    number |= (1 << 7);  
    number |= (1 << 15);  
    number |= (1 << 23);  
    number &= (1 << 31);  
    return number;  
}
```

```
int32_t bitstuff(int32_t number) {  
    number |= (1 << 7);  
    number |= (1 << 15);  
    number |= (1 << 0);  
    number |= (1 << 31);  
    return number;  
}
```

The last show case I have is less problematic than the others, but shows another nice side effect of mutation testing. Here, on the left is the original code, on the right there are two mutants with the changes highlighted in bold.

Two important things: this is not the real code from production, but its simplified version just to give you idea.

Second, I do not show the corresponding test since there was no test for this code, but the code was heavily used.

## Example #3

```
int32_t bitstuff(int32_t number) {  
    number |= (1 << 7);  
    number |= (1 << 15);  
    number |= (1 << 23);  
    number |= (1 << 31);  
    return number;  
}
```

```
int16_t fixed(int16_t number) {  
    number |= (1 << 7);  
    number |= (1 << 15);  
    return number;  
}
```

```
int32_t bitstuff(int32_t number) {  
    number |= (1 << 7);  
    number |= (1 << 15);  
    number |= (1 << 23);  
    number &= (1 << 31);  
    return number;  
}
```

```
int32_t bitstuff(int32_t number) {  
    number |= (1 << 7);  
    number |= (1 << 15);  
    number |= (1 << 0);  
    number |= (1 << 31);  
    return number;  
}
```

What happened is that there were a number of mutations: half of them were detected, while the other half not.

It turned out, that any changes to the upper 16 bits of the number did not change anything. The program was only using the lower 16 bits.

In this case we could just drop the dead code and use the int16 instead, as in the lower left function.

## Why Mutation Testing?

- Evaluates quality of a test suite
- ???
  - Incorrect test
  - Potential Vulnerability
  - Dead code

With these three examples we've seen an issue with a test. We have also seen a potential vulnerability. We have even found some dead code.



## Why Mutation Testing?

- ~~Evaluates quality of a test suite~~
- Shows semantic gaps between the test suite and the software
  - Incorrect test
  - Potential Vulnerability
  - Dead code
  - Many more things

So I'd rephrase the original, limiting definition of mutation testing with more broad and correct one - Mutation Testing helps finding semantic gaps between the test suite and the software under test.

## Brief history of mutation testing

- Invented by Richard Lipton in 1971
- Implemented by Timothy Budd in 1980
- ...
- ...
- ...
- Still niche/academia topic in 2020 (though it's slowly changing)

Now, I hope I convinced you that mutation testing brings real value let me tell you few words about the history of mutation analysis. The very first time mutation testing was mentioned in the 1971, by Richard Lipton, while the first implementation has seen the world in 1980. It was done by Timothy Budd. And now, it is 2020. I don't have time to tell you what happened to mutation testing during those 40 years, but I can definitely tell you what didn't happen - mutation testing did not became popular and didn't get as much traction as other testing techniques. I cannot know the reasons for sure, but by wild guess is that it's because of performance. So, let's look at some numbers.

## Performance

```
for_each(mutant)
```

1. Add a change
2. Compile
3. Link
4. Run tests
5. Rollback the change
6. Repeat (go to step 1)

To understand the performance implications we should first understand how mutation testing works. So, here is a brief algorithm:

- Add the change, Compile, Link, Run tests, Rollback the change, Repeat (go to step 1)

## Performance

for_each(mutant)	execution time(N) =
1. Add a change	N * (time to change)
2. Compile	+ N * (time to compile)
3. Link	+ N * (time to link)
4. Run tests	+ N * (time to run tests)
5. Rollback the change	+ N * (time to rollback)
6. Repeat (go to step 1)	

With this process in mind we can calculate the execution time for N mutants. The formula is simple, take the time each step takes and multiply it by the number of mutants, and of course sum up all the results.

# Performance

```
// test.c

int S = 1000000;
void test(int a, int b) {
    int x = 0;
    int i = S;
    while (i-- > 0) {
        x = a + b;
        x = a + b;
        x = a + b;
        // 47 lines more
    }
}

// mutant_0.c

int S = 1000000;
void test(int a, int b) {
    int x = 0;
    int i = S;
    while (i-- > 0) {
        x = a - b;
        x = a + b;
        x = a + b;
        // 47 lines more
    }
}

// mutant_1.c

int S = 1000000;
void test(int a, int b) {
    int x = 0;
    int i = S;
    while (i-- > 0) {
        x = a + b;
        x = a - b;
        x = a + b;
        // 47 lines more
    }
}
```

For the sake of numbers, I crafted an example program. You can see it on the left. What it does is just adds two numbers 50 times. In the middle and on the right you can see examples of mutations added to this program. In total, there are 50 mutants each replacing a distinct plus with a minus.

## Performance

```
baseline = compile(0.03s) + link(0.04s) + run(0.01s) = ~0.08s  
naïve execution time = mutate(0.02s) +  
    50 * compile(0.03s) +  
    50 * link(0.04s) +  
    50 * run(0.01s) = ~4.02s  
naïve slowdown = ~4.02s / 0.08s = ~50x
```

I run this code on my machine and here are the actual numbers. The important one is the baseline. This is how much time it takes to just compile, link and run the original program.

To calculate the naive execution time for 50 mutants you need to multiply the baseline by 50 and add the time needed to apply mutations.

Obviously, in this case the slowdown is around 50 times.

# Performance

```
> clang -g -fembed-bitcode test.c -o test
> mull-cxx -test-framework=CustomTest -mutators=cxx_add_to_sub test
[info] Extracting bitcode from executable (threads: 1)
[#####] 1/1. Finished in 4ms
[info] Loading bitcode files (threads: 1)
[#####] 1/1. Finished in 11ms
[info] Compiling instrumented code (threads: 1)
[#####] 1/1. Finished in 12ms
...
/tmp/sc-MkpAK7Yit/test.c:54:11: warning: Survived: Replaced + with - [cxx_add_to_sub]
    x = a + b;
      ^
/tmp/sc-MkpAK7Yit/test.c:55:11: warning: Survived: Replaced + with - [cxx_add_to_sub]
    x = a + b;
      ^
[info] Mutation score: 1%
[info] Total execution time: 526ms
```

Here is the result of running Mull against this example: total execution time is only half a second.

## Performance

baseline = compile(0.03s) + link(0.04s) + run(0.01s) = **~0.08s**

naïve execution time = mutate(0.02s) +

50 \* compile(0.03s) +

50 \* link(0.04s) +

50 \* run(0.01s) = **~4.02s**

naïve slowdown = ~4.02s / 0.08s = **~50x**

mull execution time = **~0.5s**

mull slowdown = ~0.5s / 0.08s = **~6x**

However, if we run Mull, then the execution time is around 0.5 seconds. Which slows down the original run by the factor of 6.



## Performance

baseline = compile(0.03s) + link(0.04s) + run(0.01s) = **~0.08s**

naïve execution time = mutate(0.02s) +

50 \* compile(0.03s) +

50 \* link(0.04s) +

50 \* run(0.01s) = **~4.02s**

naïve slowdown = ~4.02s / 0.08s = **~50x**

mull execution time = **~0.5s**

mull slowdown = ~0.5s / 0.08s = **~6x**

Applying Mutation Analysis On Kernel Test Suites: An Experience Report

<https://ieeexplore.ieee.org/document/7899043/>

~3500 hours!

If you think that I'm overdramatizing and exaggerating then I highly recommend reading the following paper. This is an awesome work. The authors did mutation analysis of the Linux RCU module and they used naive approach, which took roughly three and a half **thousand** hours of computational time.

# Mull Algorithm

```
#include <assert.h>

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

int main() {
    // Associative: a + b = b + a
    assert(sum(5, 10) == sum(10, 5));

    // Commutative: a + (b + c) = (a + b) + c
    assert(sum(3, sum(4, 5)) == sum(sum(3, 4), 5));

    return 0;
}
```

Now, let's get back to the very first example and see what makes Mull faster.

# Mull Algorithm

```
#include <assert.h>

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

int main() {
    // Associative: a + b = b + a
    assert(sum(5, 10) == sum(10, 5));

    // Commutative: a + (b + c) = (a + b) + c
    assert(sum(3, sum(4, 5)) == sum(sum(3, 4), 5));

    return 0;
}

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

int sum_mutant_0(int a, int b) {
    return a - b;
}

int sum_mutant_1(int a, int b) {
    return a * b;
}

int sum_mutant_2(int a, int b) {
    return a / b;
}
```

First of all, with Mull we decided to not generate a separate file for each change. Instead, we create a number of copies of the original function with one function per mutation. Besides that we also store a copy of the original function as well.

# Mull Algorithm

```
#include <assert.h>

int (*sum_ptr)(int, int) = sum_original;

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

int main() {
    // Associative: a + b = b + a
    assert(sum(5, 10) == sum(10, 5));

    // Commutative: a + (b + c) = (a + b) + c
    assert(sum(3, sum(4, 5)) == sum(sum(3, 4), 5));

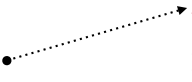
    return 0;
}

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

int sum_mutant_0(int a, int b) {
    return a - b;
}

int sum_mutant_1(int a, int b) {
    return a * b;
}

int sum_mutant_2(int a, int b) {
    return a / b;
}
```

A dotted arrow points from the variable `sum_ptr` in the line `int (*sum_ptr)(int, int) = sum_original;` to the function definition `int sum_original(int a, int b) {`.

The next step is to create a function pointer and point it to the copy of the original implementation, as you can see in the slide.

# Mull Algorithm

```
#include <assert.h>

int (*sum_ptr)(int, int) = sum_original;

int sum(int a, int b) {
    return sum_ptr(a, b);
}

int main() {
    // Associative: a + b = b + a
    assert(sum(5, 10) == sum(10, 5));

    // Commutative: a + (b + c) = (a + b) + c
    assert(sum(3, sum(4, 5)) == sum(sum(3, 4), 5));

    return 0;
}
```

```
int sum_original(int a, int b) {
    return a + b;
}

int sum_mutant_0(int a, int b) {
    return a - b;
}

int sum_mutant_1(int a, int b) {
    return a * b;
}

int sum_mutant_2(int a, int b) {
    return a / b;
}
```

And finally, replace the original implementation with an indirect call via the function pointer. Now, with this code in place we can just change the pointer to a very specific mutated function.

The question is how to do it easily and how to do it efficiently?

## Mull Algorithm

- Load program's Bitcode into memory
- Scan each function to find instructions to mutate
- Generate mutants
- Lower bitcode into machine code
- Execute each mutant via JIT engine (in a forked/isolated process)
- Report results

Mull it over: mutation testing based on LLVM

<https://arxiv.org/abs/1908.01540>

The answer is simple: just use LLVM! This is a very high-level overview of how Mull works.

As mentioned, we are trying to hide as many details as possible, so as a user you only need to build your program such that it contains Bitcode. Then feed it to Mull and you will get the results, eventually.

Under the hood, Mull loads the Bitcode into memory and starts analysis - it scans all functions and collects all instructions that can be mutated.

For each such instruction Mull creates a separate function and applies the change to the given instruction.

Once all the mutants are generated, Mull compiles the bitcode down to the machine code and prepares that machine to be executed via a JIT engine.

Then, Mull runs tests against each mutant and collects results, which are presented to the user in the end.

I'll now dive deeper into some details, but I won't be able to cover all of them. If you want to learn more - please, take a look at the corresponding paper.

# Mull Algorithm

## Find instructions to mutate

```
for (auto &module : allModules) {  
    for (auto &function : module) {  
        for (auto &instruction : function) {  
            if (canMutate(instruction)) {  
                mutate(instruction);  
            }  
        }  
    }  
}
```

Initially, we used a naïve algorithm: simply walk over each instruction in the program and collect the ones we can mutate.

The problem with this algorithm is that there are many instructions that are never executed during a test run: think of error and exception handling, or just a code that is not covered by tests at all.

# Mull Algorithm

## Find instructions to mutate

```
for (auto &module : allModules) {  
    for (auto &function : module) {  
        for (auto &instruction : function) {  
            if (canMutate(instruction)) {  
                mutate(instruction);  
            }  
        }  
    }  
}
```

```
auto coveredFunctions =  
    runTestsWithCoverage();  
for (auto &function : coveredFunctions) {  
    for (auto &instruction : function) {  
        if (canMutate(instruction)) {  
            mutate(instruction);  
        }  
    }  
}
```

We decided to use the code coverage information to only scan functions that are executed during a normal test run. It adds a certain overhead, but it also gives a nice speed up if the code coverage is not 100%. In the end, a change in the behaviour cannot be detected if it is unreachable.



# Mull Algorithm

## Generate mutants

### Source Code

a + b

### C/C++

```
entry:
    %2 = add i64 %0, %1
    ret i64 %2
```

### Swift (and Rust)

```
entry:
    %2 = tail call { i64, i1 }
        @llvm.sadd.with.overflow.i64(i64 %0, i64 %1)
    %3 = extractvalue { i64, i1 } %2, 1
    br i1 %3, label %6, label %4

ok:
    %5 = extractvalue { i64, i1 } %2, 0
    ret i64 %5

err:
    tail call void @asm.sideeffect("", "n")(i32 0)
    tail call void @llvm.trap()
    unreachable
```

Previously, I mentioned that Mull is targeting C and C++. Even though it works on the Bitcode level and can potentially be applied to any other language such as Swift or Rust, the thing is that sometimes the bitcode is way too different from the original code. Here is the simplest example I can think of. Clang turns integer addition into one instruction, while safer languages such as Swift and Rust generate many more instructions.

# Mull Algorithm

## Mutation Operators

Operator Name	Operator Semantics
cxx_add_assign_to_sub_assign	Replaces += with -=
cxx_add_to_sub	Replaces + with -
cxx_sub_assign_to_add_assign	Replaces -= with +=
cxx_sub_to_add	Replaces - with +
cxx_xor_to_or	Replaces ^ with
cxx_and_to_or	Replaces & with
cxx_or_to_and	Replaces   with &
cxx_le_to_gt	Replaces <= with >
cxx_eq_to_ne	Replaces == with !=
remove_void_function_mutator	Removes calls to a function returning void

Full List: <https://mull.readthedocs.io/en/latest/SupportedMutations.html>

Currently, Mull supports 41 mutation operator specific to C and C++. Here are some of them. Implementing the same set for other languages is possible, but is much less trivial and would require significant work.

# Mull Algorithm

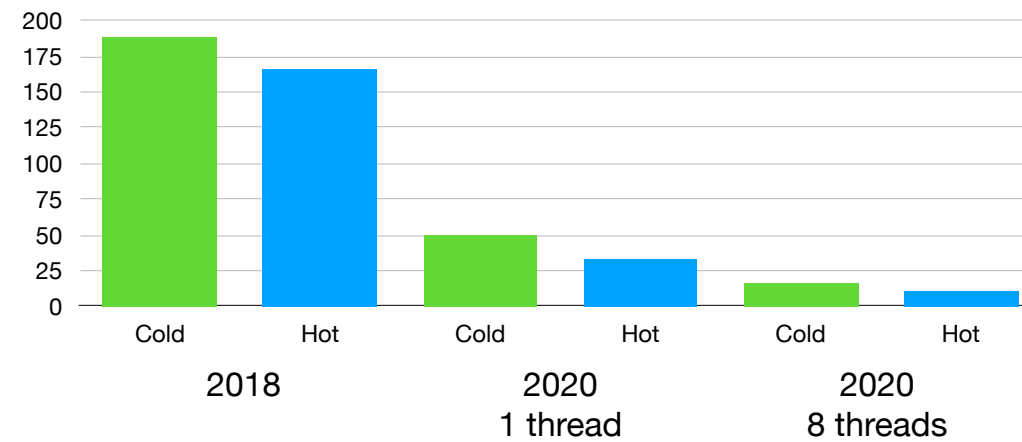
## Run mutants in forked process

- Clean state for each run
- Sandboxing
  - mutated code can crash
  - mutated code can hit deadlock/infinite loop
  - mutated code can *sometimes* crash

As promised, it is time to talk about the determinism. Each test and each mutation is run in a separate process. One of the reasons is that we need to ensure that each run happens in the clean state. The other important reason is that we cannot trust the mutants: after a change the code can hit a deadlock or be stuck in an infinite loop, or it can simply crash. To make sure mutants do not run endlessly we put a timeout and terminate the offending process after a certain threshold. Sometimes, a mutant can stuck in an infinite loop and crash because of stack overflow, the other time the same mutant can be killed because of the timeout before hitting the stack overflow.

## Real Numbers

### OpenSSL test suite



I think I gave you enough details for now. Let me conclude by showing some numbers and also show the progress in performance improvements Mull gained during the last two years. The major improvement happened when we replaced all the naïve algorithms with more sophisticated ones. The other huge improvement is obviously multithreading support.

What took around 3 minutes in 2018 today can be accomplished in less than 20 seconds. I think we can make even faster, but we will see where it goes!

## Try It

- <https://mull.readthedocs.io/en/latest/GettingStarted.html>
- <http://github.com/mull-project/mull>
- Mull it over: mutation testing based on LLVM  
<https://arxiv.org/abs/1908.01540>
- Building an LLVM-based tool: lessons learned  
<https://www.youtube.com/watch?v=Yvj4G9B6pcU>

With that being said, I can only suggest you give it a try or send patches if you hit some rough edges. If you want to learn more about the internals then I suggest you take a look at a paper and at my previous talk.

**Questions?**