"But my tests passed!"

Exploring C++ Test Suite Weaknesses with Mutation Testing

BOSCH

"I am the proof that your C++ code's intent is fragile. Try to catch me, or I'll drag your quality down into the abyss!"

About myself

- Personal Background
  - Nico Eichhorn from Stuttgart
  - Married, father of 3
  - Studied Applied Mathematics (Master Thesis at Bosch CR)

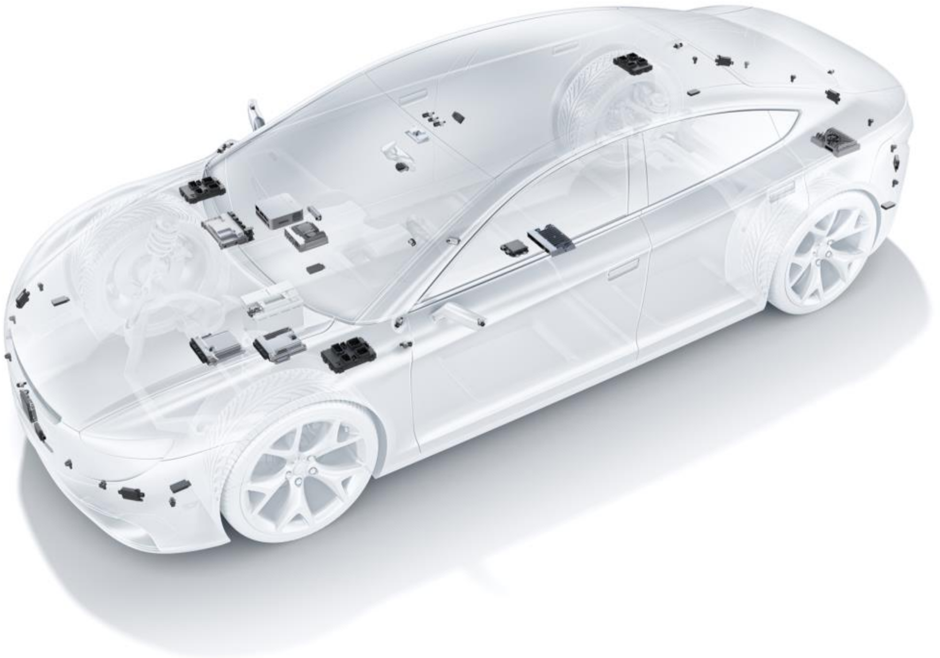
- Professional Experience
  - 12 years of C++ development
  - Visualization at High-Performance Computing Center Stuttgart
  - Automated test systems for semiconductors near Stuttgart
  - At Bosch since 2018 developing ADAS software

- Personal Motivation
  - Properly testing C++ code is crucial for robust software development
  - High coverage doesn't automatically mean your tests are effective.
  - Mutation testing can help to find an answer to" what are good tests?"

www.linkedin.com/in/nico-eichhorn-dev

What do we do at Bosch?

- Automotive software development @ Bosch
- We develop ADAS software stacks for assisted and automated driving and parking
- Part of a team which develops a central library with common functionality used by domain specific software components
- Basically, a standard template library to be used in automotive software where special safety regulations must be met
- Really excited for C++26 as it will offer more ways of making C++ safer



BOSCH

1

What is mutation testing?

What is mutation testing?
Definition & Key Concepts

- **Mutation Testing** is a software testing technique used to evaluate the quality of a test suite by introducing small changes (mutations) to the code and checking if the existing tests can detect these changes.

- **Mutants**:

  Variants of the original program created by making small modifications, such as changing operators, altering constants, or modifying control flow.

- **Mutation Operators**:

  Rules used to generate mutants. Common operators include replacing arithmetic operators, logical operators, and relational operators.

- **Test Suite Evaluation**:

  The effectiveness of a test suite is measured by its ability to detect mutants. A high-quality test suite should catch most, if not all, mutants.

BOSCH

What is mutation testing
What is the difference to fuzz testing?

- **Purpose:**
  - **Mutation Testing:** Evaluates test suite effectiveness by introducing small code changes (mutations) and checking if existing tests detect them.
  - **Fuzz Testing:** Discovers program vulnerabilities, crashes, or unexpected behavior by feeding semi-random, malformed, or unexpected input data.

- **Process:**
  - **Mutation Testing:** Modifies the *code*. Reruns the *existing* test suite to detect these changes.
  - **Fuzz Testing:** Generates and feeds *input data* to the program to observe behavior.

- **Outcome:**
  - **Mutation Testing:** Ensures test robustness, highlighting gaps in test coverage or assert statements.
  - **Fuzz Testing:** Identifies program defects, security vulnerabilities, or unexpected behavior caused by anomalous inputs.

BOSCH

What is mutation testing?
Process

1. **Generate Mutants**: Create multiple versions of the original code with small changes.

2. **Run Tests**: Execute the existing test suite against each mutant.

3. **Analyze Results**:
   - **Killed Mutants**: Mutants that cause tests to fail, indicating the test suite is effective.
   - **Surviving Mutants**: Mutants that pass all tests, suggesting gaps in the test suite.

BOSCH

What is mutation testing?
Benefits & Challenges

- **Improves Test Quality**: Identifies weaknesses in the test suite, ensuring it covers more edge cases and scenarios.
- **Increases Confidence**: Provides a higher level of assurance that the code is robust and well-tested.
- **Encourages Better Testing Practices**: Promotes the development of more comprehensive and effective tests.

- **Performance Overhead**: Running tests on multiple mutants can be time-consuming and resource-intensive.
- **Complexity**: Managing and analyzing many mutants can be complex.
- **Tool Support**: Requires specialized tools to automate the generation and testing of mutants.

BOSCH

What is mutation testing?
Tools

- Dextool Mutate
- MuCPP
- mull
- Universal Mutator

- Other examples for different languages:
  - Java: PIT/Pitest
  - Python: MutPy/Cosmic Ray
  - Javascript/Typescript: Stryker
  - Rust: Mutagen

- Mull:
  - Uses clang for parsing and AST generation
  - Leverages clang's tooling for code analysis and transformation
  - Generates mutants by applying these operators to the AST.
  - Compiles all mutants using clang into a binary.
  - Ensures mutants are syntactically correct and executable.
  - Executes the test suite against each mutant.
  - Isolates test runs to ensure accurate results.

BOSCH

2

Finding Test Suite Weaknesses

mutation testing 32-bit floating point type addition

Case study
What are we testing?

- A simple algorithm which emulates addition for 32-bit floating point values as defined in IEEE754

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

= 0x40490FDB

$1 \times 2^{1} \times 1.5707964 = 3.1415927$

- Special cases:
  - NaN: Invalid operation e.g. sqrt(-1)

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

= 0x7FC00000

$1 \times 2^{128} \times 1.5 = \text{NaN}$

  - Infinity: e.g. division by zero

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

= 0x7F800000

$1 \times 2^{128} \times 1 = \text{Infinity}$

  - Largest positive subnormal

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

= 0x007FFFFF

$1 \times 2^{-126} \times 0.9999999 = 1.1754942e{-}38$

- Simplification: subnormal results are treated as 0

Created with **https://evanw.github.io/float-toy/**

BOSCH

Case study
What are we testing

- Inputs and output are 32-bit unsigned integers representing 32-bit floats
- The algorithm is implemented as following:
  1. Special case filtering (NaN / infinity)
  2. Operand preparation + exponent alignment
  3. Signed mantissa combine (add/subtract + cancellation)
  4. Normalization (overflow / underflow handling)
  5. Repack and finalize (including overflow to infinity)

BOSCH

Case study
Preparing

- Install from packages provided by mull project
  - https://github.com/mull-project/mull
- Create a config
- All targets need to be compiled with the following flags:
  - `-fpass-plugin=/usr/lib/mull-ir-frontend-18`
    - specifies a plugin for the pass manager, the mull frontend in this case
    - Will "infect" the code with mutants
  - -grecord-command-line
    - to record the command line options used during compilation in the debug information

```yaml
mutators:
  - cxx_bitwise
  - cxx_comparison
  - cxx_arithmetic
  - cxx_boundary
  - cxx_calls
excludePaths:
  - gtest
  - gmock
  - test
```


BOSCH

Case Study
Available Mutators

- There are numerous groups to choose from, such as:
  - **`cxx_arithmetic`**: Modifies fundamental mathematical calculations.
    - Example: `cxx_add_to_sub` (Changes `a + b` to `a - b`)
  - **`cxx_comparison`**: Alters how values are compared, including boundary conditions.
    - Example: `cxx_eq_to_ne` (Changes `a == b` to `a != b`)
  - **`cxx_assignment`**: Targets various forms of value assignment and initialization.
    - Example: `cxx_add_assign_to_sub_assign` (Changes `a += b` to `a -= b`)
  - **`cxx_increment` / `cxx_decrement`**: Flips the direction of unary increment/decrement operations.
    - Example: `cxx_pre_inc_to_pre_dec` (Changes `++x` to `--x`)
  - **`cxx_calls`**: Manipulates function calls, often removing their effects.
    - Example: `cxx_remove_void_call` (Removes a call to a `void` function)

Case study
Running mull

```
mull-runner-18 test/mainTest
[info] Using config /workspace/mull.yml
[warning] Could not find dynamic library: libstdc++.so.6
[warning] Could not find dynamic library: libm.so.6
[warning] Could not find dynamic library: libgcc_s.so.1
[warning] Could not find dynamic library: libc.so.6
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 209ms
[info] Extracting coverage information (threads: 1)
       [################################] 1/1. Finished in 4ms
[info] Filter mutants (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 180ms
[info] Running mutants (threads: 16)
       [################################] 56/56. Finished in 4228ms
[info] Survived mutants (7/56):
/workspace/inc/utils.hpp:103:27: warning: Survived: Replaced != with == [cxx_ne_to_eq]
```

… some output …

```
                          ^
[info] Mutation score: 87%
[info] Total execution time: 4626ms
[info] Surviving mutants: 7
```

- 7 mutants survived…

Case study
Analyzing results

```
/workspace/inc/utils.hpp:103:27: warning: Survived: Replaced != with == [cxx_ne_to_eq]
    if(expA == 0 && mantA != 0)
                          ^
/workspace/inc/utils.hpp:112:27: warning: Survived: Replaced != with == [cxx_ne_to_eq]
    if(expB == 0 && mantB != 0)
                          ^
/workspace/inc/utils.hpp:124:13: warning: Survived: Replaced > with >= [cxx_gt_to_ge]
    if(expA > expB)
            ^
/workspace/inc/utils.hpp:129:18: warning: Survived: Replaced > with >= [cxx_gt_to_ge]
    else if(expB > expA)
                 ^
/workspace/inc/utils.hpp:178:22: warning: Survived: Replaced >= with > [cxx_ge_to_gt]
    while(resultMant >= 0x1000000) // Handle overflow
                     ^
/workspace/inc/utils.hpp:181:22: warning: Survived: Replaced < with <= [cxx_lt_to_le]
        if(resultExp < 255) // Prevent overflow to infinity
                     ^
/workspace/inc/utils.hpp:188:22: warning: Survived: Replaced > with >= [cxx_gt_to_ge]
        if(resultExp > 0) // Only normalize if exponent is non-zero
                     ^
```

- Every line represents a survived mutant
- A potential opportunity for improvement in tests

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
workspace/inc/utils.hpp	100.00% (1/1)	100.00% (120/120)	100.00% (73/73)	96.55% (56/58)
Totals	**100.00% (1/1)**	**100.00% (120/120)**	**100.00% (73/73)**	**96.55% (56/58)**

Generated by llvm-cov -- llvm version 18.1.3

Case study
Analyzing results

```
[info] Survived mutants (7/56):
/workspace/inc/utils.hpp:103:27: warning: Survived: Replaced != with == [cxx_ne_to_eq]
    if(expA == 0 && mantA != 0)
                          ^
/workspace/inc/utils.hpp:112:27: warning: Survived: Replaced != with == [cxx_ne_to_eq]
    if(expB == 0 && mantB != 0)
                          ^
```

- What does this mean?
  - With this mutation zero is treated as subnormal and exponent is set to 1
  - Later this change is "corrected" when exponents and mantissas are aligned for addition
  - If we test zero added to zero, no correction will occur, and the test would fail

```
TEST(EmulateFloatAdditionTest, ZeroPlusZero)
{
    // Test zero + zero to kill mantA != 0 mutant
    uint32_t zero1 = floatToBits(0.0F);
    uint32_t zero2 = floatToBits(0.0F);
    uint32_t resultZero = emulateFloatAddition(zero1, zero2);
    EXPECT_FLOAT_EQ(bitsToFloat(resultZero), 0.F);
    EXPECT_EQ(resultZero, 0x00000000); // Must be exactly zero, not corrupted
}
```

Case study
Analyzing results

```
/workspace/inc/utils.hpp:124:13: warning: Survived: Replaced > with >= [cxx_gt_to_ge]
    if(expA > expB)
            ^
/workspace/inc/utils.hpp:129:18: warning: Survived: Replaced > with >= [cxx_gt_to_ge]
    else if(expB > expA)
                 ^
```

- changes >= to > code block for alignment of mantissas.
- Equality means bit-shift by 0 bits
- Both branches can do this
- We cannot handle this mutant

```
if(expA > expB)
{
    mantB >>= (expA - expB);
    expB = expA;
}
else if(expB > expA)
{
    mantA >>= (expB - expA);
    expA = expB;
}
```

Case study

Analyzing results

```
/workspace/inc/utils.hpp:178:22: warning: Survived: Replaced >= with > [cxx_ge_to_gt]
    while(resultMant >= 0x1000000) // Handle overflow
                     ^
```

- Mantissa is not normalized anymore

- Very specific bit-pattern to test

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 0x7F000000

$1 \times 2^{127} \times 1 = 1.7014118e+38$

  - ➢ Triggers mantissa overflow after implicit addition of leading 1 for mantissa

```
TEST(EmulateFloatAdditionTest, ExponentExactly254)
{
    uint32_t a = 0x7F000000U; // 1.7014118E+38
    uint32_t b = 0x7F000000U; // Same

    uint32_t result = emulateFloatAddition(a, b);

    EXPECT_EQ(result, 0x7F800000U);
}
```

Case study
Analyzing results

```
/workspace/inc/utils.hpp:181:22: warning: Survived: Replaced < with <= [cxx_lt_to_le]
        if(resultExp < 255) // Prevent overflow to infinity
                     ^
```

- Again, normalization for mantissa overflow
- 8-Bit exponent
  - 0xFE: max value for regular numbers
  - 0xFF: ±NaN / ±inf handled beforehand

- Refactoring needed to handle this mutant

```
if(resultExp <= 254)
```

- But is this better?

BOSCH

Case study
Demo time

https://github.com/nicoeich87/MutationTesting

3

Conclusion

BOSCH

Conclusion
Summary

- Mutation testing can help to improve test quality
  - Found possible improvements for emulated 32-bit float addition test
  - Almost perfect coverage, yet we found that some (edge-case) test cases missing

- Can easily added to existing tests
  - Proper configuration is important to avoid unwanted noise

- Analysis can be very time consuming
  - AI can help to understand and fix findings, however not always a quicker way.

BOSCH

Conclusion
Final thoughts

- Can be used for individual testing / examination of tests and algorithms.
  - Great for code with a high cyclomatic complexity
- Easy to setup and use but some caveats
  - More time consuming as analysis and execution is more complex
  - Can only mutate runtime code (no consteval)
- For usage in CI analysis of git diffs can be a useful addition
  - Several ways of integrating in pipelines
- If you want to really improve your tests you should try mutation testing

BOSCH

Thank you

BOSCH