Mutation Testing at Scale

@giorgionatili

$ whoami

- Engineering leader at Amazon (Kindle rendering team)
- Organizer of Droidcon Boston (and maybe Seattle)
- Organizer of SwiftFest Boston and Atlanta
- Meetups and community enthusiast
- Lead of the System Architecture & Design learning track in Amazon


Disambiguate
Software Quality

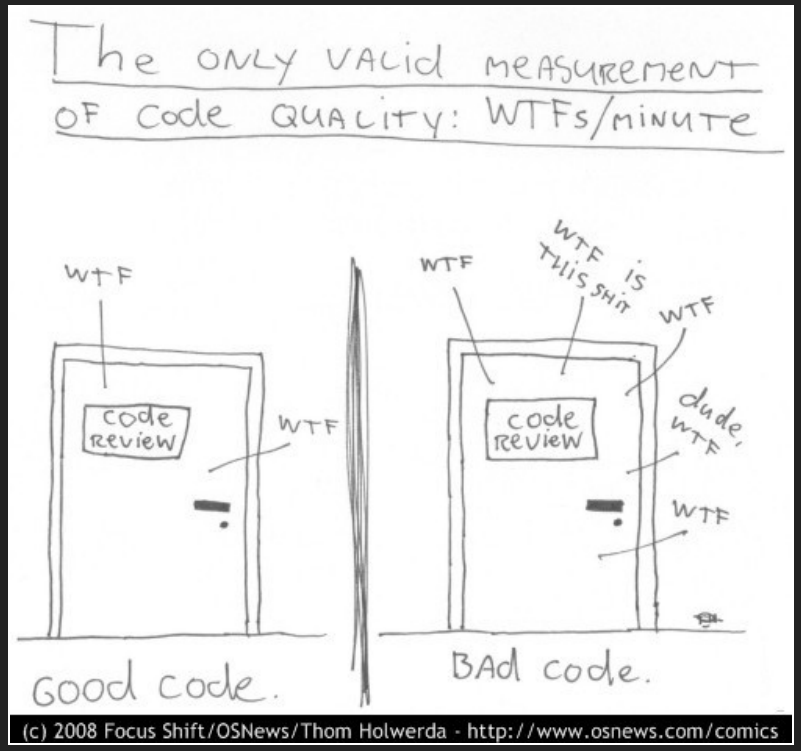
Quality starts with clean code

```
1 int d; // elapsed time in days
```

VS

```
1 int elapsedTimeInDays;
```

Measuring Code Quality



Clean code
is testable

Speaker notes

* If you understand what the code is doing, you can also tets its intentions and create safe boundaries in which you can play (i.e., safe modifications)

Adding tests clean the code

Speaker notes

* Adding tests should not be complicated, when it's the case it means that the code is convoluted and too complicated
* Don't forget the difference between complex (i.e., interesting) and complicated (i.e., most of the times wrong)

Rigorous TDD

Speaker notes

* Write the test first, all the times
* Write the test that proves the bug exists
* Never let a quick fix driving your code (i.e., stop designing)

Tests as First-Class Citizens

Anatomy of a Good Test

- Self-descriptive
- Simple
- SOLID

Speaker notes

Evert simple test should satisfy at least two principles:
S — Single responsibility principle
O — Open for extensions, but closed for modification (when you start to modify a test by adding new assertions, most of the time you are modifying the source code instead of extending it)

More sophisticated tests should satisfy also other principles:

What to test?

```
1 namespace stringutil {
2
3    std::string tail(const std::string& word) {
4       if (word.length() == 0) return "";
5       return word.substr(1);
6    }
7 }
```

Speaker notes

What conditions do you want to test here?

- Given a string, when it's empty, then the tail is empty as well
- Given a string, when it contains only one character, then the tail is empty
- Given a string, when it contains more than one character, then the tail should contain the rest of the string after the head

3S Based Tests

```
1  TEST(AString, AllTheLettersAfterTheHeadAsShlouldBeTheTail) {
2     ASSERT_THAT(tail("xyz"), Eq("yz"));
3  }
4
5  TEST(AString, TheTailOfAnEmptyStringShouldBeEmpty) {
6     ASSERT_THAT(tail(""), Eq(""));
7  }
8
9  TEST(AString, TheTailOfASingleCharacterStringShouldBeEmpty) {
10    ASSERT_THAT(tail("X"), Eq(""));
11 }
```

A Good Test Suite

- Reliable
- Accurate
- Fast

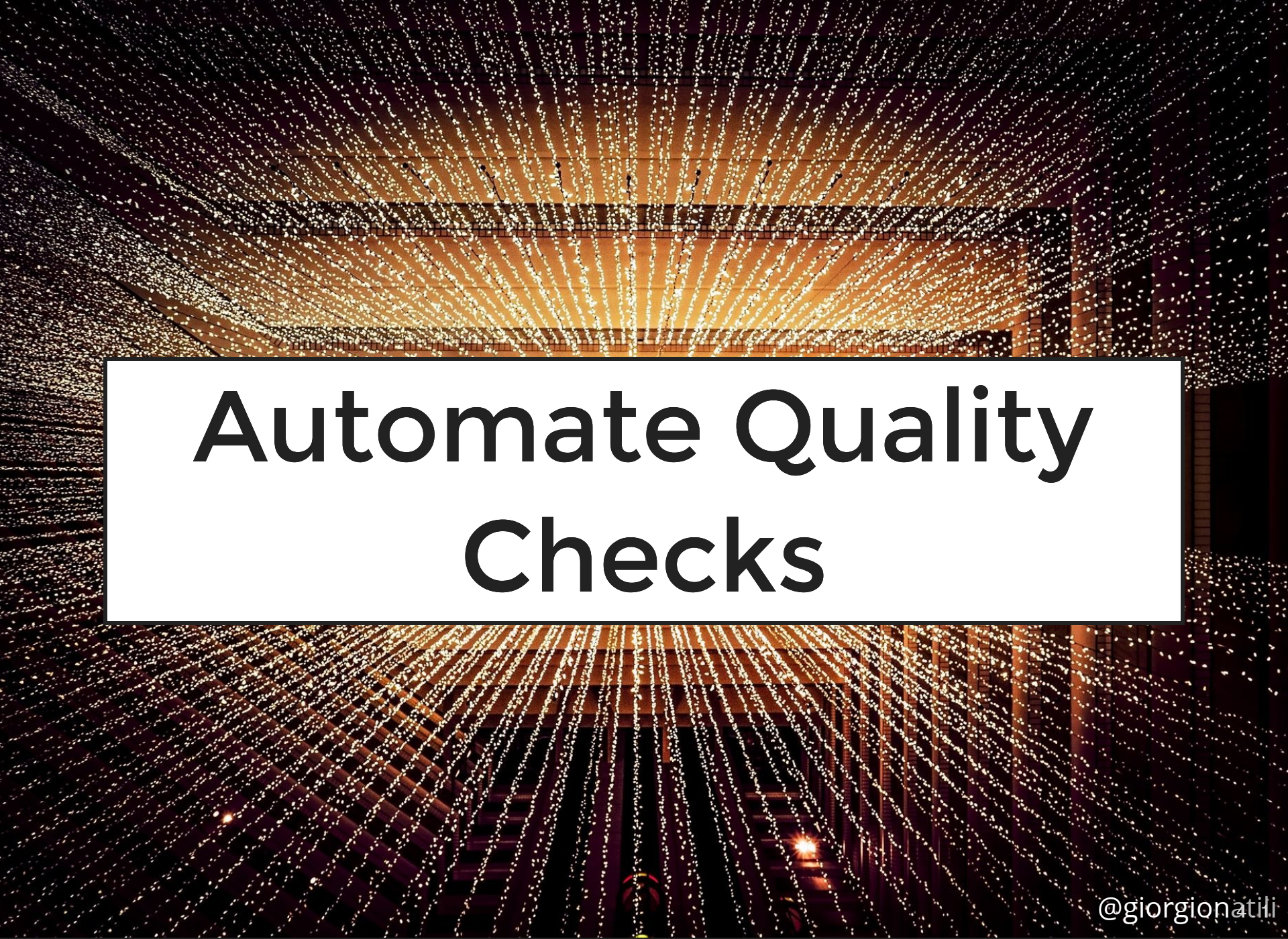
Speaker notes

* In software development, a test suite, less commonly known as a 'validation suite', is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors
* A test suite often contains detailed instructions or goals for each collection of test cases and information on the system configuration to be used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests

Potential Test Suite Quality Metrics

- Line coverage
- Tests reliability
- Execution speed

Which are the right metrics?


Automate Quality Checks
@giorgionatili

Is beneficial adding tests?

Speaker notes

If you add more tests, does this mean your tests suite is stronger? Or is it just causing an exaggerated overhead on adding functionality?

It depends on the kind and quality of tests; when you and your team are copying and pasting tests to cover new features you should start to dive deep on why and how to improve it.

What is the right test coverage?

Speaker notes

There are some pitfalls in pursuing a random coverage index:

* It's not possible to determine if the areas under test are in the critical path of your software (should we say product?)
* Until a real bug is not found, it's not easy to be sure to test the boundaries of your software
* Specs and real use cases are often pretty different, how many times do you implement an API that then was used

What is the test coverage?

```
1  TEST(AString, AllTheLettersAfterTheHeadAsShlouldBeTheTail) {
2     ASSERT_THAT(tail("xyz"), Eq("yz"));
3  }
4
5  TEST(AString, TheTailOfAnEmptyStringShouldBeEmpty) {
6     ASSERT_THAT(tail(""), Eq(""));
7  }
8
9  TEST(AString, TheTailOfASingleCharacterStringShouldBeEmpty) {
10    tail("X");
11 }
```

The Oracle Problem

Speaker notes

* Coverage misses one important aspect: The Oracle Problem
* A test oracle is an entity that decides whether a test case passed or failed.

Specified
These oracles are typically associated with formalized approaches to software modeling and software code construction.

Different Approach

- Learning from earlier mistakes to prevent them from happening again
- Simulate earlier mistakes and see whether the resulting defects gets discovered

Fault Based Testing

Not Always Black or White!

Speaker notes

Because it's not always black or white


Fuzzing All the Things
@giorgionatili

Goals

- Measure the degree to which a system, component, or function can work with an invalid or stressful input
- Deviate from the normal expected input of a program to analyze the consequences

Input Validation

```
1 bool checkEvenOdd(int num){
2     return num % 2 == 0 ? true : false;
3 }
```

```
1 bool isDigit(char *c_array){
2     for (int k = 0; k < strlen(c_array); k++) {
3         if ((int)c_array[k]<(int)'0' ||
4             (int)c_array[k]>(int)'9') {
5             return false;
6         }
7     }
8     return true;
9 }
```

Yet

Pretty

Open

The Heart bleed bug

heartbleed.com

Speaker notes

* The heartbeat extension provides a way to test and keep alive secure communication links without the need to renegotiate the connection each time
* The affected versions of OpenSSL allocate a memory buffer for the message to be returned based on the length field in the requesting message, without regard to the actual size of that message's payload
* As a result, a malicious user can steal the servers' private keys and users' session cookies and passwords

Benefits

- Early bugs finding
- Discover security issues
- Discover fragile areas of the codebase

Approaches

- Dumb fuzzers (mutation)
- Intelligent fuzzers (generation)

Speaker notes

* Dumb fuzzers (mutation)
    ** Here, it’s all about mutating the existing input values (blindly) and that’s why it is known as “dumb” fuzzers, as in lacking understanding of the format/structure of the data
* Intelligent fuzzers (generation)
    ** In contrast to Dumb Fuzzers, here an understanding of the file format/protocol is very important; this approach is about "generating" the inputs from scratch based on the specification/format.

Drawbacks

- Fuzz testing alone cannot provide a complete picture of an overall security threat or bugs
- Fuzz testing can detect only simple faults or threats
- To perform effectively, it will require significant time
- Setting a boundary value condition with random inputs is very problematic

Tools

- Fuzzing Frameworks
  - Boofuz
  - BDFuzz
- Mutational Fuzzers (alter existing data samples to create new test data)
  - AFL / libFuzzer
  - Radamsa

Speaker notes

libFuzzer can be checked out from LLVM's Subversion repository and built using their directions. You supply a test-driver as a function called LLVMFuzzerTestOneInput with C linkage. The result is a standalone program that exercises the code inside that function. It uses some Clang compiler-supplied instrumentation, via the -fsanitize-coverage option, to monitor which paths are exercised, so gcc is not an option.

AFL is a standalone tool that uses binary rewriting to instrument the code being tested. It supplies wrapper

Fuzzing Doesn't Listen

Speaker notes

Fuzzing doesn't cover complex scenarios, mutation listen because it uses the tests as a driver for creating mutants.


Mutation Testing
@giorgionatili

Unexpected Program Mutations

What Is It?

- Mutation testing evaluates the quality of existing software tests
- The idea is to modify (i.e, mutate) code covered by tests in a small way and check whether the existing test set detects or rejects the change

Mutation Testing Framework

- Alter source code in one very small way
- Run unit tests
- Record if any tests fail

Speaker notes

A mutation testing framework will go through this process:
* Alter source code in one very small way
* Run unit tests
* Record if any tests fail

Mutants

- Each transformation results in a new program, called mutant, that differs from the original
- Detecting and rejecting such a modification by the existing tests is denoted as killing a mutant

Speaker notes

* The process of creating a mutant from the original program is called mutagenesis

Killing Mutants

Metrics

- Test suite effectiveness is measured by its ability to detect those mutants
- The mutation score is the ratio of killed mutants to the total number of mutants

Speaker notes

* Mutation score is, therefore, the measurement of the test suite effectiveness

What About Test Coverage?

Speaker notes

* Full coverage alone testifies nothing about the quality of the underlying tests! It is pretty useless from the quality assurance point of view
* Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests

" This is where mutation testing comes into play!

Different Mutations

- Statement mutation
- Value mutation
- Decision mutation

Speaker notes

In Software Engineering, Mutation testing could be fundamentally categorized into 3 types – statement mutation, decision mutation, and value mutation.

* Statement Mutation — cut and pastes a part of a code of which the outcome may be a removal of some lines
* Value Mutation — values of primary parameters are modified
* Decision Mutation — control statements are changed

Statement Mutation

```
1  // Initial code:
2  if(a < b) {
3    c = 10;
4   } else {
5    c = 20;
6  }
7
8  // Changed code:
9  if(a < b) {
10   d = 10;
11  } else {
12   d = 20;
13 }
```

Speaker notes

* In statement mutations, a statement is deleted or it is replaced by some other statement

Value Mutation

```
1  // Initial code:
2  int mod = 1000000007;
3  int a = 12345678;
4  int b = 98765432;
5  int c = (a + b) % mod;
6
7  // Mutated code:
8  int mod = 1007;
9  int a = 12345678;
10 int b = 98765432;
11 int c = (a + b) % mod;
```

Speaker notes

* Basically a small value is changed to a larger value or a larger value is changed to a smaller value. In this testing basically constants are changed

Decision Mutation

```
1  // Initial code:
2  if(a < b) {
3   c = 10;
4  } else {
5   c = 20;
6  }
7
8  // Mutated code:
9  if(a > b) {
10  c = 10;
11  } else {
12  c = 20;
13 }
```

Speaker notes

* In decision mutations, the logical or arithmetic operators are changed to detect errors in the program

Dedicated mutation operators

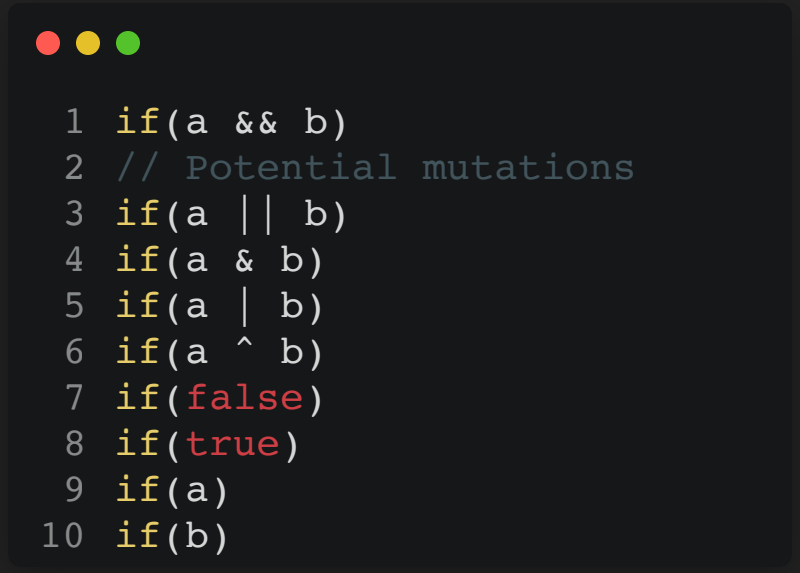
Arithmetic Operator Replacement

```
1  int greatestCommonDenominator(int x, int y) {
2
3      int tmp;
4      while(y != 0) {
5          tmp = x % y; // The % operator can be replaced
6          x = y;       // with +,-,*,/,%,**
7          y = tmp;
8      }
9      return x;
10 }
```

Relational Operator Replacement

```
1  int greatestCommonDenominator(int x, int y) {
2
3      int tmp;
4      while(y != 0) {     // The != operator can be
5          tmp = x % y;    // replaced by <,>,<=,>=,=, !=
6          x = y;
7          y = tmp;
8      }
9      return x;
10 }
```

Conditional Operator Replacement



```
1  if(a && b)
2  // Potential mutations
3  if(a || b)
4  if(a & b)
5  if(a | b)
6  if(a ^ b)
7  if(false)
8  if(true)
9  if(a)
10 if(b)
```

Many Others

- Assignment Operator Replacement
- Unary Operator Insertion
- Scalar Variable Replacement
- Absolute Value Insertion

Speaker notes

Object Oriented Muation:

* AMC - Access Modifier Change
* HVD - Hiding Variable Deletion
* HVI - Hiding Variable Insertion
* OMD - Overriding Method Deletion

Mutation Testing

- Identifies areas of code that are not tested properly
- Identifies hidden defects that can’t be detected using other testing methods
- Assesses the quality of the test cases
- Assesses error propagation in the program

Speaker notes

Infinite loops and runtime errors can happen with a mutant and this can be useful to build the code to better manage errors during the normal workflow.

Mutation Testing
+
Mutation Analysis

Speaker notes

* In a nutshell:
    * Mutation analysis: Assessing the quality of a test suite
        * Mutation analysis inserts systematic faults (mutations) into the source code under test producing mutants of the original code and judges the effectiveness of the test suite by its ability to detect those faults
    * Mutation testing: Improving the test suite using mutants
        * Mutants resemble real-world bugs, and that the test suite effectiveness in detecting mutants is correlated to

A Lot of Data

Mutation testing based on LLVM

Supported Languages

Java, JVM

PHP

Rust

C, C++

Closure

Python

Javascript

Scala

Ruby

Swift

C#

Let's Focus On

Java, JVM

PHP

Rust

C, C++

Closure

Python

Javascript

Scala

Ruby

Swift

C#

LLVM

Speaker notes

* The LLVM Project is a collection of a modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project
* LLVM makes it easier to not only create new languages but to enhance the development of existing ones
* C is sometimes described as a portable, high-level assembly language, LLVM’s IR was designed from the beginning to be a portable assembly

Available Tools

- Dextool Mutate, plugin based on Dextool
- MuCPP, based on source code mutants generation
- Mull, an LLVM-based tool with a focus on C and C++
- CCMutator, based on higher-order mutation operators implemented as opt passes on LLVM IR
- Xemu, based on QEMU software emulator

Speaker notes

* Dextool Mutate https://github.com/joakim-brannstrom/dextool/blob/master/plugin/mutate/README.md (Can continue from where a testing session was interrupted, allows multiple instances to be run in parallel)
* MuCPP https://ucase.uca.es/mucpp/download.html (Test suite independent, but it seems outdated)
* MULL https://arxiv.org/pdf/1908.01540.pdf (Supports multiple languages, works well with GoogleTest)
* CCMutator https://cpb-us-e1.wpmucdn.com/sites.usc.edu/dist/c/321/files/2019/03/Kusano13CCmutator-1dgag3o.pdf (7 years that the repo is not updated)

What is Mull

- An open-source tool for mutation testing based on LLVM
- An extendable tool to analyze the effectiveness of your test suite
- A command-line tool that produces a SQLite database or an HTML report of the tested program

Speaker notes

* It works with LLVM IR, a low-level intermediate representation, to perform mutations, and uses LLVM JIT for just-in-time compilation; for this reason, Mull is:
    * Language independent (any language that supports LLVM IR -> C, C++, Rust, Swift)
    * Fast (fine-grained control over compilation and execution of the program and its mutants)
* Direct manipulation of LLVM IR allows Mull to do less work to generate mutations: only modified fragments of IR code are recompiled, and this results in faster processing of mutated programs

How to Run Mull

```
mull-cxx -test-framework=GoogleTest -mutators=conditional \
         -reporters=Elements -report-dir=./report \
         -report-name=MULL-TEST-ONE \
         -workers=4 -compdb-path compile_commands \
         -disable-cache=0 \
         ./bin/core-test
```

Speaker notes

Steps: before running Mull:

* git clone https://github.com/mull-project/mull.git --recursive
* cd mull
* mkdir build.dir && cd $_
* cmake -DPATH_TO_LLVM=path/to/llvm DCMAKE_CXX_FLAGS=-D_GLIBCXX_USE_CXX11_ABI=0 ..

Why Mull

- Efficiency in generating a mutation
- Support for dry-run mode
- Effective sandbox model
- Support for failing fast

Speaker notes

* Direct manipulation of LLVM IR allows Mull to do less work to generate mutations: only modified fragments of IR code are recompiled, and this results in faster processing of mutated programs
* When in dry-run mode, Mull collects information about mutants but doesn’t run the tests against them
* Mutations can impact the code behavior and make the program crash, timeout or exit prematurely; Mull uses a parent/child process isolation
* Mull has an option to decrease the number of test runs that is _fail fast mode_

Supported Mutators

- Mathematical
- Conditional negator
- Remove void function
- Replace call
- Scalar value replacement
- Many others! :)

Speaker notes

Supported Mutators (also known as operators)

* Math: Add, Sub, Mul, Div; this group of operators performs mutations of basic arithmetic operators such as "+" to "-", "-" to "+", "*" to "/", "/" to "*", and so on
* Conditional negator, this group of operators negate a condition such as "lt" to "gt", "eq" to "ne", and so on
* Remove void function mutator, it removes the calls to a function returning void from LLVM IR code

Explore the Mutators

```
$ ./mull-cxx --help
```

```
--mutators=<value>                                  - Choose mutators:
    =all                                              -   default, experimental
    =arithmetic                                       -   cxx_arithmetic_add_to_sub, cxx_arithmetic_add_assign_to_su
    =bitwise                                          -   cxx_bitwise_lshift_to_rshift, cxx_bitwise_lshift_assign_to
    =conditional                                      -   and_or_replacement_mutator, negate_mutator, conditionals_b
    =conditionals_boundary_mutator                    -   cxx_relational_le_to_lt, cxx_relational_lt_to_le, cxx_rela
    =constant                                         -   scalar_value_mutator
    =cxx                                              -   conditionals_boundary_mutator, negate_relational, arithmet
    =default                                          -   cxx_arithmetic_add_to_sub, negate_mutator, remove_void_fun
    =experimental                                     -   and_or_replacement_mutator, numbers, replace_call_mutator,
    =functions                                        -   replace_call_mutator, remove_void_function_mutator
    =math                                             -   cxx_arithmetic_add_to_sub, cxx_arithmetic_sub_to_add, cxx_
    =negate_relational                                -   cxx_relational_gt_to_le, cxx_relational_ge_to_lt, cxx_rela
    =numbers                                          -   cxx_number_init_const, cxx_number_assign_const
    =and_or_replacement_mutator                       -   Replaces && with ||, || with &&
    =cxx_arithmetic_add_assign_to_sub_assign          -   Replaces += with -=
    =cxx_arithmetic_add_to_sub                        -   Replaces + with -
    =cxx_arithmetic_div_assign_to_mul_assign          -   Replaces /= with *=
    =cxx_arithmetic_div_to_mul                            Replaces / with *
```

Mull's Approach

- Mutations can be done either at a high level (i.e., source code) or at a lower level (i.e., bitcode)
- Mull applies mutations at a lower level because:
  - The same engine can be used to support any LLVM-based language
  - The execution time for each mutation is lower

Speaker notes

* Mutating at the LLVM link
  ** Same mutation in IR syntax
  ** Use JIT to compile and link all the small mutation
  ** Faster but not all the mutation at this level have representation in code
* Mutate at binary
  ** Even faster feedback loop

Under the Hood

- Loads LLVM bitcode into memory
- Inserts instrumentation code into each function
- Compiles instrumented LLVM bitcode to machine code
- Prepares the machine code for execution by the LLVM JIT engine
- At an IR code level, it finds the matching tests
- Runs each test using the LLVM JIT engine and collects code coverage information

Speaker notes

* Step 1: Mull loads LLVM Bitcode into memory
* Step 2: Mull inserts instrumentation code into each function. This code is used to collect code coverage information.
* Step 3: Mull compiles instrumented LLVM Bitcode to machine code and prepares the machine code for execution by LLVM JIT engine.
* Step 4: In the LLVM IR code Mull finds the tests according to a test framework specified in the configuration file

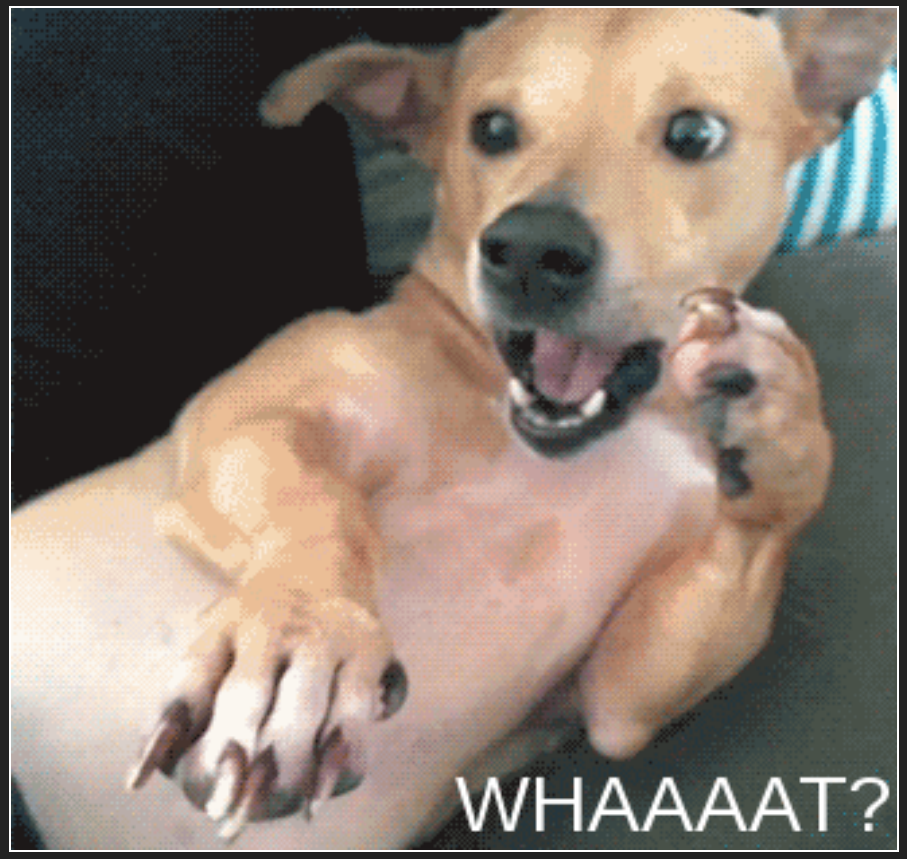
Drawbacks

- Compiling with bitode enabled is straightforward for a small project but painful for big projects
- Mutating the bitcode generates noise because not all the mutations have a representation in code
- Some mutations generate the same behavior
- Mutation testing is time-consuming and requires brain power
- It is not a solution for black-box testing

Speaker notes

* Unfortunately, lots of the low-level 3P code has custom build systems (e.g. ICU, Boost).
* Faster but not all the mutation at this level have representation in code; more noise because not all the mutations have a representation in code
* Not all mutations are interesting because some will result in the exact same behavior (i.e., Equivalent Mutations)

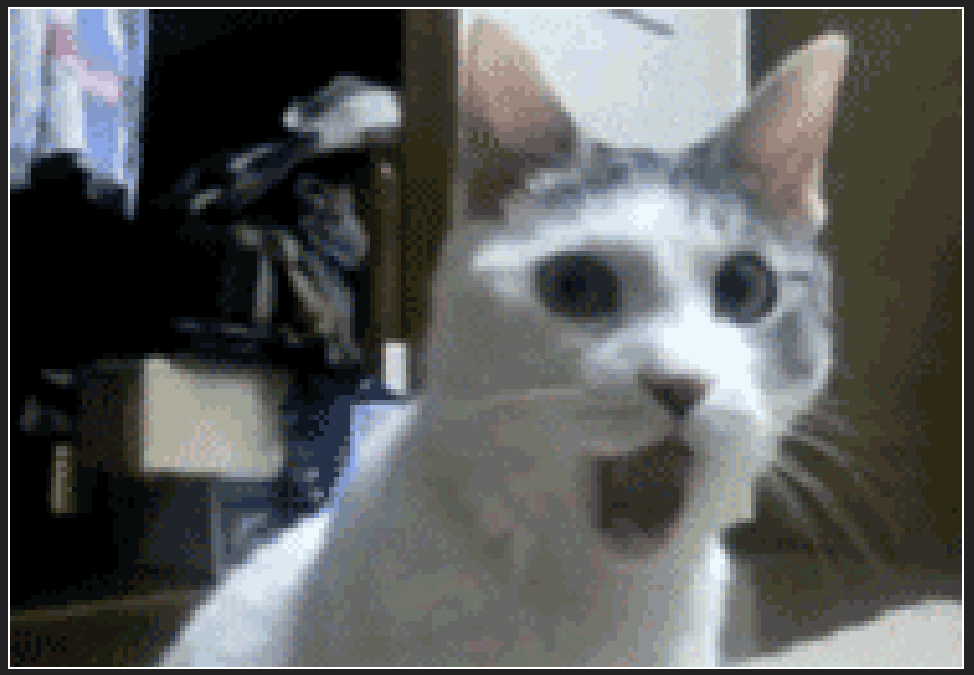
What?!?



Then Why?

- To identify potential areas of improvement
- To find bugs behind the usual human interaction
- To optimize error handling strategies
- To asses the quality and health status of the codebase
- To estimate the remaining unknowns bugs of a program

How is it possible?



Reports and Metrics

Discover

Inspect

Improve

Generate Reports

```
mull-cxx -test-framework=GoogleTest -mutators=math \
         -reporters=Elements -report-dir=./report -report-name=TEST \
         -workers=4 -compdb-path compile_cmd.json -disable-cache=0 \
         -compilation-flags="\
            -isystem /opt/clang+llvm-9.0.0/include/c++/v1 \
            -isystem /opt/clang+llvm-9.0.0/lib/clang/9.0.0/include \
            -isystem /usr/include" \
         ./bin/core-test
```

Speaker notes

{fmt} is an open-source formatting library for C++. It can be used as a safe and fast alternative to (s)printf and iostreams.

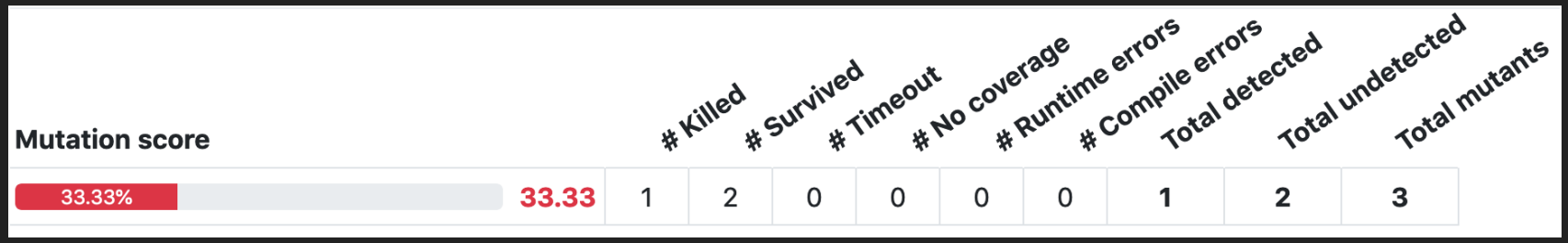
Math mutators -> cxx_arithmetic_add_to_sub, cxx_arithmetic_sub_to_add, cxx_arithmetic_mul_to_div, cxx_arithmetic_div_to_mul

Exploring Logs

```
Loading bitcode files (threads: 4): 4/4. Finished in 267ms.
Compiling instrumented code (threads: 4): 4/4. Finished in 11ms.
Loading dynamic libraries (threads: 1): 1/1. Finished in 0ms.
Searching tests (threads: 1): 1/1. Finished in 2ms.
Preparing original test run (threads: 1): 1/1. Finished in 145ms.
Running original tests (threads: 4): 30/30. Finished in 187ms.
Applying function filter: no debug info (threads: 4): 3496/3496. Finished in 14ms.
Applying function filter: file path (threads: 4): 3313/3313. Finished in 22ms.
Instruction selection (threads: 4): 3313/3313. Finished in 23ms.
Searching mutants across functions (threads: 4): 3313/3313. Finished in 369ms.
Applying filter: no debug info (threads: 4): 12355/12355. Finished in 12ms.
Applying filter: file path (threads: 4): 12355/12355. Finished in 35ms.
Applying filter: junk (threads: 4): 12355/12355. Finished in 3657ms.
Prepare mutations (threads: 1): 1/1. Finished in 0ms.
Cloning functions for mutation (threads: 4): 4/4. Finished in 769ms.
Removing original functions (threads: 4): 4/4. Finished in 194ms.
Redirect mutated functions (threads: 4): 4/4. Finished in 11ms.
Applying mutations (threads: 1): 409/409. Finished in 11ms.
Compiling original code (threads: 4): 4/4. Finished in 3625ms.
Running mutants (threads: 4): 409/409. Finished in 4586ms.
```

Mutation Score

(fmt / include / fmt / core.h with math mutators)



Mutation score		# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
33.33%	33.33	1	2	0	0	0	0	1	2	3

Survived Mutant

(fmt / include / fmt / core.h with math mutators)

```
// Advances the begin iterator to ``it``.
FMT_CONSTEXPR void advance_to(iterator it) {
  format_str_.remove_prefix(internal::to_unsigned(cxx_arithmetic_sub_to_add + ~~it - begin()~~));
}

// Returns the next argument index.
FMT_CONSTEXPR int next_arg_id() {
  if (next_arg_id_ >= 0) return next_arg_id_++;
```

Replaced - with +

👽 Survived

Speaker notes

Math mutators cxx_arithmetic_add_to_sub, cxx_arithmetic_sub_to_add, cxx_arithmetic_mul_to_div, cxx_arithmetic_div_to_mul

Time Constraints

```
Compiling instrumented code (threads: 4): 4/4. Finished in 4612ms.
Loading dynamic libraries (threads: 1): 1/1. Finished in 0ms.
Searching tests (threads: 1): 1/1. Finished in 1ms.
Preparing original test run (threads: 1): 1/1. Finished in 86ms.
Running original tests (threads: 4): 30/30. Finished in 203ms.
Applying function filter: no debug info (threads: 4): 3496/3496. Finished in 15ms.
Applying function filter: file path (threads: 4): 3313/3313. Finished in 23ms.
Instruction selection (threads: 4): 3313/3313. Finished in 21ms.
Searching mutants across functions (threads: 4): 3313/3313. Finished in 608ms.
Applying filter: no debug info (threads: 4): 20586/20586. Finished in 15ms.
Applying filter: file path (threads: 4): 20586/20586. Finished in 58ms.
Applying filter: junk (threads: 4): 20586/20586. Finished in 3969ms.
Prepare mutations (threads: 1): 1/1. Finished in 1ms.
Cloning functions for mutation (threads: 4): 4/4. Finished in 1040ms.
Removing original functions (threads: 4): 4/4. Finished in 204ms.
Redirect mutated functions (threads: 4): 4/4. Finished in 13ms.
Applying mutations (threads: 1): 446/446. Finished in 10ms.
Compiling original code (threads: 4): 4/4. Finished in 3808ms.
Running mutants (threads: 4): 446/446. Finished in 5704ms.

Total execution time: 21046ms
```

Speaker notes

Running all the mutators with the cache disabled (i.e., a first-run) on a relatively small package takes more than 20 seconds.

Estimate Remaining Bugs

Switching Perspectives

- Count how many open bugs are in your backlog
- Label or categorize 30% of them
- Run *mull*, then categorize and count the bugs
- Calculate the ratio between categorized and not categorized bugs

Existing Data

total = 300 known bugs

labeled = 100 categorized bugs

labeledFound = 30 existing bugs discovered with mutation

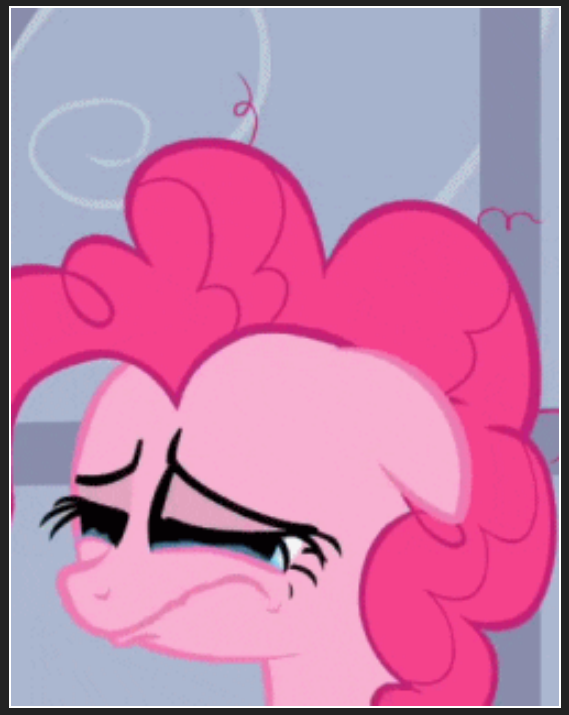
found = 100 total bugs discovered with mutation

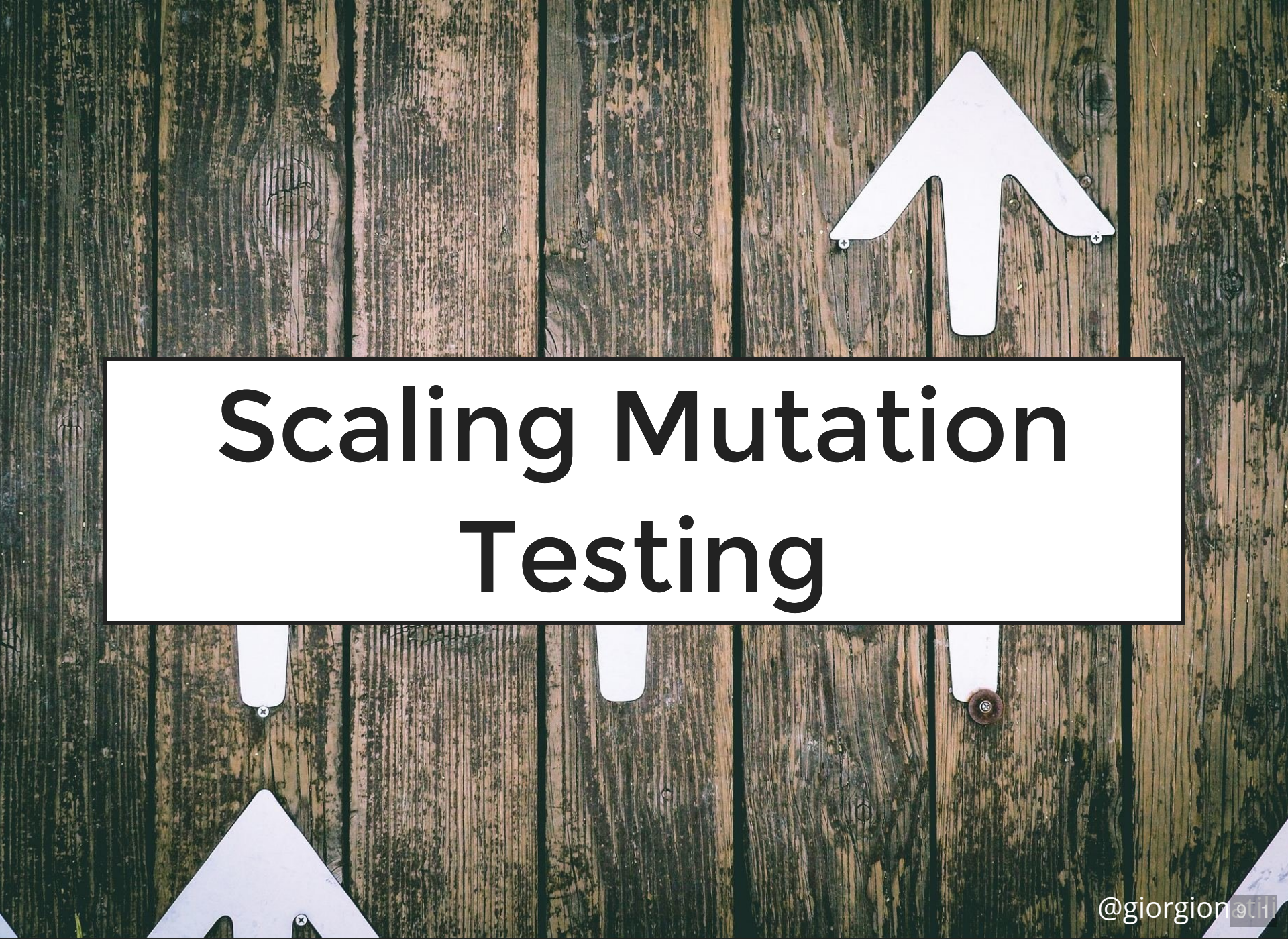
Simple Ratio

$$\frac{\text{labeled}}{\textbf{unknown}} = \frac{\text{labeledFound}}{\text{notLabeledFound}}$$

unknown > 200 potential unknown bugs

Don't Panic




Scaling Mutation Testing
@giorgionatili

Cultural Changes

- Code quality is essential to release a successful product
- It's possible to objectively evaluate code quality
- Automating quality checks is keen for effective teams
- Tests are code and should be implemented with the same criteria

Technical Challenges

- Everyone worked with obsolete compilers
- Integrate the checks in your build tools
- Minimize the junk in your data
- Support every operating system

Compilers Outdated

- Compilers can be updated
- Software architecture can simplify compilers update
- Updates are like a fresh start

Pipeline Integration

- Build infrastructure can integrate any tool
- Be thoughtful on *when* trigger mutation testing
- Analyze your data early in the process and often
- Modularize your pipeline

Dev Environment

- Invest time to simplify the usage of the tools
- Be inclusive, support all the dev platforms
- Write exhaustive documentation

Report Analysis

- Review the data in isolation and share your finding
- Collect the findings and learn from them
- Implement a data model to learn from errors

Speaker notes

* Review the report
* Collect the Equivalent Mutations and Junk mutations to learn from them

Get ready for a new challenge


Final Remarks

Speaker notes

* Fun fact, you got exposed to more than 20 new acronyms and you survived
* Extreme mutation: Extreme mutation is another mutation testing strategy to simplify and to increase mutation speed. It characterizes itself by replacing the whole method logic by a nullable block: in java, we would have no code on void methods, a simple return null; statement on methods returning objects, or returning some constants.
    * a method is a good level of abstraction to reason about the code and the test suite;
    * extreme mutation generates much fewer mutants than the default/classic strategy;

Terminology

- A *fault* is an erroneous part of a program
- A *mutation* is a fault that introduced in a program
- A *mutant* is a program created from the original one with a potential failure
- A *variant* is a program that shows a deviation at runtime from the original program
- A *redundant fault* is a duplicated fault

Ubiquity

Java, JVM

PHP

Rust

C, C++

Closure

Python

Javascript

Scala

Ruby

Swift

C#

Bitcode and Bytecode

- Same same but different (JVM instructions are stack-oriented, whereas LLVM bitcode is not)
- LLVM bitcode is closer to machine-level code, but isn't bound by a particular architecture

Speaker notes

The biggest difference between JVM bytecode and LLVM bitcode is that JVM instructions are stack-oriented, whereas LLVM bitcode is not. This means that rather than loading values into registers, JVM bytecode loads values onto a stack and computes values from there. I believe that an advantage of this is that the compiler doesn't have to allocate registers, but I'm not sure.

LLVM bitcode is closer to machine-level code but isn't bound by a particular architecture. For instance, I think that

Resources

- Fuzzing github.com/secfigo/Awesome-Fuzzing
- LLVM command line guide llvm.org/docs/CommandGuide/
- Mutation Testing github.com/theofidry/awesome-mutation-testing
- Rahul Gopinath papers rahul.gopinath.org/publications/

Speaker notes

Rahul Gopinath

Thank You!

@giorgionatili