

# Mutation testing

Guest lecture on the course  
"Development of reliable, safe and secure software"

May 26, 2021

# Intro

Alex Denisov

- LLVM/compiler enthusiast
- Software Engineer at GitHub
- [alex@lowlevelbits.org](mailto:alex@lowlevelbits.org) / [https://twitter.com/1101\\_debian](https://twitter.com/1101_debian)

Stanislav Pankevich

- 7 startups
- Last 3 years in a German NewSpace company
- Software development, software quality, development tools

# Agenda

- Normal software vs critical software
- Overview of the testing methods
- Mutation testing method
- 10-minute break
- Demonstration: Mull tutorial with some examples
- Mull mutation testing tool
- Lessons learned
- Q/A

## Goals:

- Share a snapshot of our industry experience
- Share a passion for open-source development
  - Insight into the LLVM infrastructure and software quality tool development

"Normal" and "critical" software

# Interview question: "normal" and "critical" software

There's an aircraft. And there is software to be designed and implemented.

- You have unlimited time to develop and verify the software.
- You are going to fly on that aircraft.
- What can you do to make sure you can fly it safe?

# Interview question: "normal" and "critical" software (2)

Here are some answers:

- Lot of testing, all of it. Full test coverage. 100% MC/DC. 100% path coverage. Inputs are known to produce correct outputs.
- Understanding the context where software is supposed to operate.
  - Electronics/hardware failures. What is the supervising code?
- Interfaces. Depending on the criticality of the software:
  - If not critical, separate from critical.
  - If critical, an unlimited study can be made on the design that takes existing and new software into account.
- The question does not indicate how many attempts. So, a lot of iterative testing can be assumed with an insight into telemetry data.
- Put a great team to work. Various skills, expertise, perspectives.
- Give developers great tools to do the job.
- Rewrite everything to Rust (existing software and new software).
- Simulator to test all aspects of the system behavior.
- Formal methods, model-based development and testing

# Critical software (informally)

## Risks:

- Safety of people
- Damage to environment
- Expensive missions and projects, loss of equipment
- Financial loss
- Bottom line: Unacceptable loss

## Attributes:

- Mistakes that are impossible or very hard or expensive to fix
- Very often embedded software (but not always)
- Development governed by standards (process!, requirements!)
- Higher degrees of development and testing thoroughness

# Criticality categories

- Slightly different variations across industries and standards
  - Differences in number of categories, their boundaries, and their names
- A - life of people is threatened
- B - catastrophic, loss of a mission
- C - major damage
- D - minor, negligible, easily correctable errors
- *"For software not directly involved in the operation of the system, but rather in its testing (such as simulators or software validation facilities), it is important that its criticality is also determined based on the indirect consequence of its failures on the system." (ECSS)*



# ECSS criticality categories (ECSS-Q-ST-30C)

Category	Consequences	Effects
A	<b>Catastrophic</b> (Loss of life, life-threatening or permanently disabling injury or occupational illness)	<ul style="list-style-type: none"><li>• Loss of system</li><li>• Loss of an interfacing manned flight system</li><li>• Loss of launch site facilities</li><li>• Severe detrimental environmental effects</li></ul>
B	<b>Critical</b> (Loss of mission, temporarily disabling but not life-threatening injury, or temporary occupational illness)	<ul style="list-style-type: none"><li>• Major damage to an interfacing flight system</li><li>• Major damage to ground facilities</li><li>• Major damage to public or private property</li><li>• Major detrimental environmental effects</li></ul>
C	<b>Major</b>	Major mission degradation
D	<b>Minor or Negligible</b>	Minor mission degradation or any other effect

# Business side of things

- "Test everything" => "How much we can produce to an acceptable quality while being commercially efficient"
- "Number of FTEs" and "Rate per Hour" instead of "code coverage"
- Schedules, "Everyone is busy"
- Startups - lightweight quality processes
  - Getting to MVP without any or less quality
- Full V&V + IVV processes are less accessible to smaller companies
- "Safety argument"
  - We have tried hard to achieve safety. This is what we did.
- A system with heritage is preferred to custom development
- Systems engineering and software systems engineering
  - Then software development and testing
- Lots of testing => lots of tradeoffs

# NASA's Program/Project Types

Criteria	Type A	Type B	Type C	Type D	Type E	Type F
Description of the Types of Mission	Human Space Flight or Very Large Science/ Robotic Missions	Non-Human Space Flight or Science/Robotic Missions	Small Science or Robotic Missions	Smaller Science or Technology Missions (ISS payload)	Suborbital or Aircraft or Large Ground based Missions	Aircraft or Ground based technology demonstrations
Priority (Criticality to Agency Strategic Plan) and Acceptable Risk Level	High priority, very low (minimized) risk	High priority, low risk	Medium priority, medium risk	Low priority, high risk	Low priority, high risk	Low to very low priority, high risk
National Significance	Very high	High	Medium	Medium to Low	Low	Very Low
Complexity	Very high to high	High to Medium	Medium to Low	Medium to Low	Low	Low to Very Low
Mission Lifetime (Primary Baseline Mission)	Long. >5 years	Medium. 2–5 years	Short. <2 years	Short. <2 years	N/A	N/A
Cost Guidance (estimate LCC)	High (greater than ~\$1B)	High to Medium (~\$500M–\$1B)	Medium to Low (~\$100M–\$500M)	Low (~\$50M–\$100M)	(~\$10–50M)	(less than \$10–15M)

Source: NASA Systems Engineering Handbook (NASA SP-2016-6105 Rev2)

# NASA's Program/Project Types (2)

Launch Constraints	Critical	Medium	Few	Few to none	Few to none	N/A
Alternative Research Opportunities or Re-flight Opportunities	No alternative or re-flight opportunities	Few or no alternative or re-flight opportunities	Some or few alternative or re-flight opportunities	Significant alternative or re-flight opportunities	Significant alternative or re-flight opportunities	Significant alternative or re-flight opportunities
Achievement of Mission Success Criteria	All practical measures are taken to achieve minimum risk to mission success. The highest assurance standards are used.	Stringent assurance standards with only minor compromises in application to maintain a low risk to mission success.	Medium risk of not achieving mission success may be acceptable. Reduced assurance standards are permitted.	Medium or significant risk of not achieving mission success is permitted. Minimal assurance standards are permitted.	Significant risk of not achieving mission success is permitted. Minimal assurance standards are permitted.	Significant risk of not achieving mission success is permitted. Minimal assurance standards are permitted.
Examples	HST, Cassini, JIMO, JWST, MPCV, SLS, ISS	MER, MRO, Discovery payloads, ISS Facility Class payloads, Attached ISS payloads	ESSP, Explorer payloads, MIDES, ISS complex subrack payloads, PA-1, ARES 1-X, MEDLI, CLARREO, SAGE III, Calipso	SPARTAN, GAS Can, technology demonstrators, simple ISS, express middeck and subrack payloads, SMEX, MISSE-X, EV-2	IRVE-2, IRVE-3, HiFIRE, HyBoLT, ALHAT, STORRM, Earth Venture I	DAWNair, InFlame, Research, technology demonstrations

# Effort ladder - ECSS deltas example

**Critical software** - software of criticality category A, B or C. (Therefore category D represents non-critical software.)

~**180** requirements (D) => ~**230** requirements (A)

	Class D	Class C	Class B	Class A
Modeling, model-based, formal methods	-	-	+	+
Depth of analysis	-	+	++	+++
Independent V&V	-	-	+	+
Testing thoroughness (documentation and traceability)	-	+	++	+++
Documentation overhead	+	++	+++	+++
Test coverage targets (statement / decision / MC/DC)	+ (Negotiable)	++ (Negotiable)	+++ 100% (MC/DC - Negotiable)	+++ 100%

# Summary: Critical software development process

- Testing activities are part of a development lifecycle
- Many influencing factors: industry, criticality, type of project, budgets, etc.
- If you are writing tests full time, it likely means that someone (you?) has prioritized this activity within a larger context

# Testing methods

# Verification and Validation?

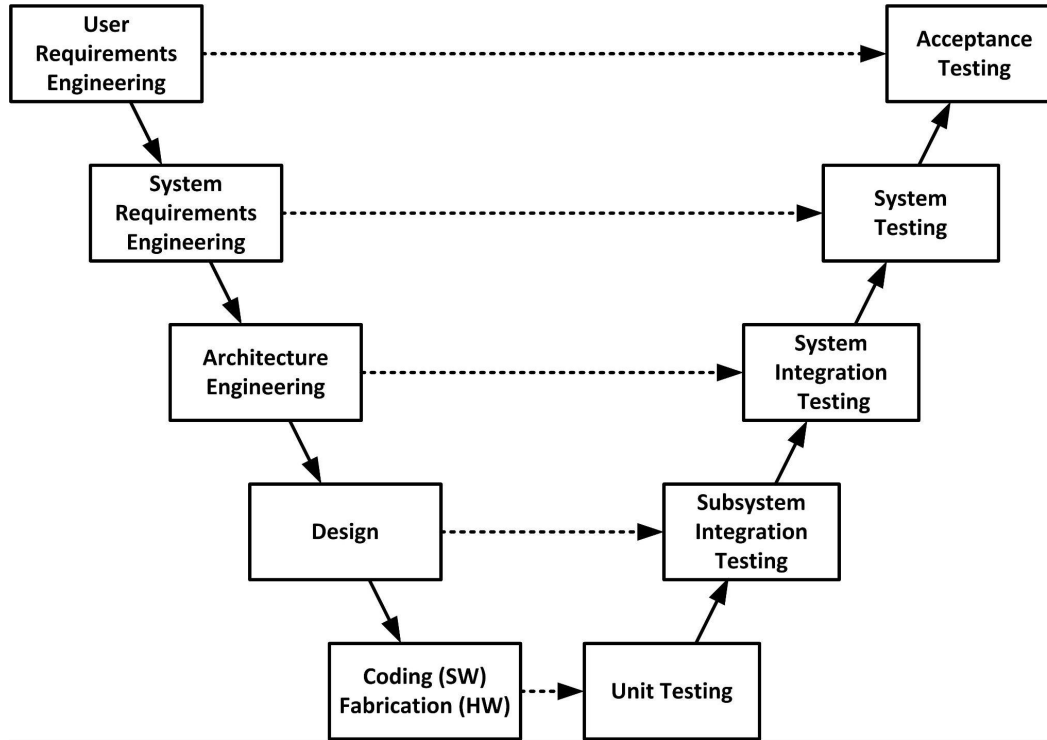
- Validation: "<software> process to confirm that the requirements baseline functions and performances are correctly and completely implemented in the final product." (ECSS)
- Verification: "<software> process to confirm that adequate specifications and inputs exist for any activity, and that the outputs of the activities are correct and consistent with the specifications and input" (ECSS)
- Requirements <-> Specifications (What vs How)
- Real-life example: Buying shoes
  - Lack of validation: You are happy with the shoes, but after a few days they don't fit and hurt.
  - Lack of verification: You are happy with the shoes, but after a few days, due to a rainy weather, cracks appear.
- Overlap and mutual influence between V&V activities



# Testing levels

- Unit testing
  - Sometimes called "coverage testing"
  - Different understanding of "unit":
    - Granularity of a single C++ class
    - Granularity of a translation unit, i.e. a C/C++ file with the headers inlined
  - The best possible control over test coverage on this level
- Integration testing
  - Very often: "functional testing"
  - Testing low-level software interfacing with hardware
  - Higher levels of integration also "component testing"
- System testing
  - Example: Hardware-in-the-Loop simulations in the Flight Software Simulation and Validation facilities
- Acceptance testing
  - Example: Pre-flight checkout using umbilical connection

# V model and testing levels



# Short overview of testing and verification methods

- Code review (inspection)
- Linters
  - Code readability (typos, naming conventions), maintainability metrics (number of lines in a file, in a function, ...)
  - Coding style
- Static analysis
  - Coding standard (MISRA, BARR-C, JPL, Power of 10, etc.)
  - More insights from tools using abstract interpretation
- Dynamic analysis
  - Sanitizers (LLVM Address, Thread, Undefined Behavior Sanitizers)
  - Symbolic execution, fuzzing
- Testing and code coverage
  - Unit, integration, validation, system / acceptance testing
- Formal methods
  - Model checking, formal proofs
- Analysis and simulations
- Model-based development and testing
  - Automatic code generation
- Risk analysis/assessment, hazard analysis

# Some observations about testing activities

- There is no a single testing method that is enough
- Inherent quality is cheaper than doing quality after the development is done
  - Test-Driven Development, early static analysis and adherence to a coding standard
- A code with 100% MC/DC coverage might still be doing the wrong job
  - Great reading: "Nancy G. Leveson. Engineering a Safer World: Systems Thinking Applied to Safety."
- Two verification steps
  - Code does what it should do according to the requirements (top-down requirements-based testing)
  - Code does not do anything else (bottom-up structural coverage testing)
    - Unintended functionality
    - Side effects
    - Finding gaps in requirements (missing, incomplete or inconsistent requirements)
- The best initial investment into testing:
  - At least one integration test for every software component (nominal course or "green" case)
    - Optionally: at least one test for a known or most probable (or easiest) failure or "red" case
  - Substantially improves further development and maintenance
  - It is easy to write more tests one there is at least one
  - The same initial investment into unit testing: at least one green case, optionally one red.

# Code coverage ladder

- Statement coverage
- Branch coverage
- Condition coverage
- MC/DC coverage
- Multiple condition coverage
- Mutation coverage?

# Statement coverage

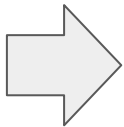
- All statements (lines) have to be executed
- Number of statements executed / total number of statements
- The most basic type of coverage
- 100% statement coverage is a good achievement

```
if (condition) {  
    doSomething();  
}  
  
return a + b;
```

# Branch and decision coverage

- Branch: all true/false branches must be visited
  - (i.e. branch coverage  $\geq$  statement coverage)
- Also called decision coverage
- Not only if statements but also assignments with boolean expressions
  - `bool d = a && (b || c);`

```
if (condition) {  
    doSomething();  
}
```

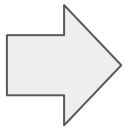


100% code coverage but the Else branch is not covered

## Branch and decision coverage (2)

- Branch: all true/false branches must be visited
  - (i.e. branch coverage  $\geq$  statement coverage)
- Also called decision coverage
- Not only if statements but also assignments with boolean expressions
  - `bool d = a && (b || c);`

```
if (condition) {  
    doSomething();  
}
```



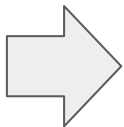
```
if (!condition) {  
    return;  
}  
doSomething();
```



# Condition coverage

- All boolean expressions must be evaluated to both true and false
- Not equivalent to branch coverage

```
if (a || b) {  
    doSomething();  
}  
else {  
    doSomething();  
}
```



(a = true, b = false) and (a = false, b = true) satisfies condition coverage but not branch coverage  
(Else branch is not taken)

# MC/DC coverage

- Mandatory in number of industries for higher criticality categories
- Condition coverage + decision coverage + independence requirement
  - Every point of entry and exit in the program has been invoked at least once
  - Every condition in a decision in the program has taken all possible outcomes at least once
  - Every decision in the program has taken all possible outcomes at least once
  - Each condition in a decision has been shown to independently affect that decision's outcome.
    - A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.

# MC/DC coverage (2)

- Developed by Chilenski and Miller
- Structural coverage testing method
- Is a measure of the adequacy of requirements-based testing
  - Undocumented implementation in software
  - Separate job: "Verification Analyst"
- An effort/cost optimization method
  - Achieve comparable error detection probability compared to full testing
  - $N + 1$  test cases for a decision with  $N$  inputs instead of  $2^N$  for (full) Multiple Condition Coverage
- 3 variations: Unique-Cause, Unique-Cause+Masking, Masking
- Focus on logic (Boolean expressions)
  - Analysis of truth tables and logical gates
  - Knowledge of the manual procedure is must-have even when testing supported by tools
- Known to detect:
  - Operator errors:  $(A \text{ and } B)$  vs  $(A \text{ or } B)$
  - Operand errors:  $(A \text{ and } B)$  vs  $(A \text{ and } C)$
  - Grouping errors:  $(A \text{ and } (B \text{ or } C))$  vs  $((A \text{ and } B) \text{ or } C)$

# MC/DC challenge

- GitHub: No mature tool for MC/DC
- Challenge: Open-source tool for MC/DC coverage based on LLVM
- LLVM API for Clang AST and LLVM IR should be enough
- Tutorials / papers:
  - A practical approach to modified condition/decision coverage
  - An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion
  - A Practical Tutorial on Modified Condition/ Decision Coverage
- Some pointers:
  - <https://github.com/Armin-Montigny/MCDC>
  - <https://github.com/ttsiodras/condition-decision-mcdc>

# Mutation testing method

# Methods based on mutation

- Fuzzers - mutation of input to discover vulnerabilities
  - libFuzzer, <https://llvm.org/docs/LibFuzzer.html>
- Fault injection
  - Injecting memory errors to simulate the effects of space radiation environment
- Mutation testing
- Genetic programming, automatic software repair

# Mutation testing method

- Systematic injection of faults into software
- Also known as Mutation Analysis
- Invented by Richard Lipton in 1971
- Implemented by Timothy Budd in 1980
- Very stable branch of research in academia
- Competent programmer hypothesis
  - Programmers write good code in general but introduce small mistakes from time to time
- Coupling effect
  - Simple smaller faults can couple and emerge into more serious faults

# Mutation testing algorithm (pseudocode)

```
run_tests(program, test_suite); // ensure tests pass
mutants = generate_mutations(program);
killed_mutants = 0;
for (mutant in mutants) {
    mutant = mutate(program);
    test_result = run_tests(mutant, test_suite);
    if (test_result == Failed) {
        report_killed_mutant(mutant, test);
        killed_mutants++;
    } else {
        report_failed_mutant(mutant, test);
    }
}
mutation_score = killed_mutants / mutants.size() * 100;
```

**Killed mutant** - the test is good: it is able to detect the mutation.

**Survived mutant** - the test is not able to detect mutation. Either this test needs improvement or more tests have to be written to kill survived mutants.

## Mutation example:

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



# Typical mutations

Arithmetic:

- a - b

+ a + b

Bitwise:

- a << b

+ a >> b

Remove void call:

- callToVoidFunction();

Logical:

- if (a && b) {

+ if (a || b) {

Unary:

- x--;

+ x++;

Replace call:

- ... = callToFunction();

+ ... = 42;

Relational:

- if (a > b) {

+ if (a >= b) {

# Mutation coverage

- Mutation Score =  $(\# \text{ of killed mutants}) / (\# \text{ of mutants}) * 100$
- The goal is 100% mutation score
- Relative metric, depends on a chosen mutation set
- Studies have shown that there are minimal sets of essential mutations
- Mutation coverage is at least as powerful as branch coverage
- "As it is actually able to detect whether each statement is meaningfully tested, mutation testing is the gold standard against which all other types of coverage are measured." (<https://pitest.org/>)

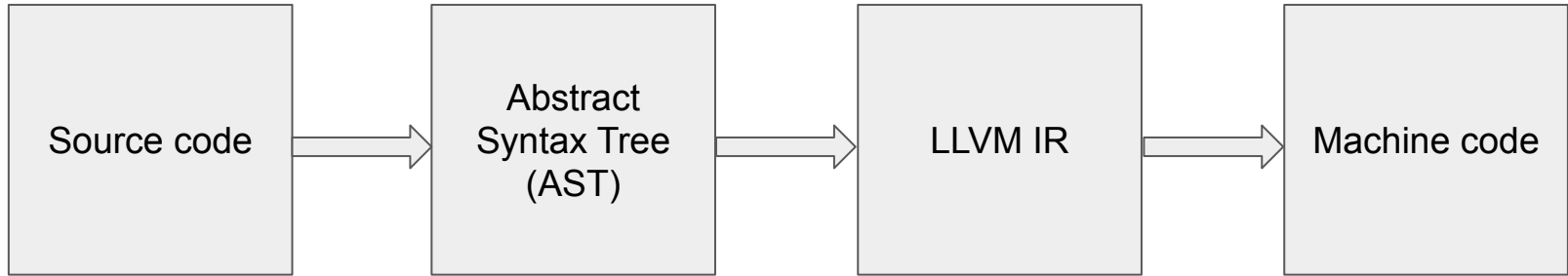
# Mutation testing tools: Many implementations

- Which layers to mutate:
  - Mutate source code
  - Mutate AST
  - Mutate Intermediate Representation (LLVM bitcode, Java bytecode)
- How mutants are generated
  - Compile separate programs (1 mutant - 1 program)
  - Metamutant or hypermutant (all mutants in a single program)
    - "Mutant schemata"
- How mutants are launched
  - Separate executable
  - Interpretation (for example, LLVM JIT)
- List of tools: <https://github.com/theofidry/awesome-mutation-testing>

# What is LLVM

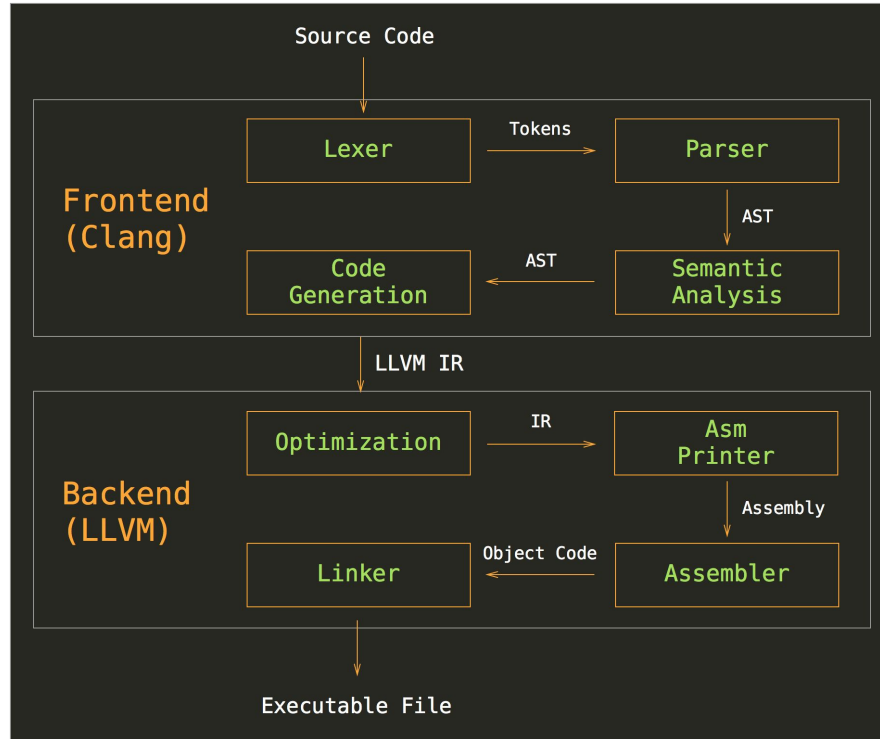
- "The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.", <https://llvm.org/>
- Frontends: Clang (C/C++ compiler), Swift, Rust, Fortran, ...
- Backends: X86, PowerPC, ARM, SPARC, ...
- Intermediate language LLVM Intermediate Representation (IR)
  - Bridge between programming languages and different hardware platforms
- Examples:
  - Writing your own static analyzer using Clang API
  - Writing your compiler optimizations
  - Write your own mutation testing tool :)

# Compilation pipeline (simplified)



Note: Some languages have their own intermediate languages between the AST and LLVM IR

# Compilation process: Clang example



# Simple code example: Clang AST representation

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

```
| -FunctionDecl 0x7fe550043d60 <line:5:1, line:7:1> line:5:5 used sum 'int (int, int)'  
| | -ParmVarDecl 0x7fe550043c00 <col:9, col:13> col:13 used a 'int'  
| | -ParmVarDecl 0x7fe550043c80 <col:16, col:20> col:20 used b 'int'  
| ` -CompoundStmt 0x7fe550043eb0 <col:23, line:7:1>  
|   ` -ReturnStmt 0x7fe550043ea0 <line:6:3, col:14>  
|     ` -BinaryOperator 0x7fe550043e80 <col:10, col:14> 'int' '+'  
|       | -ImplicitCastExpr 0x7fe550043e50 <col:10> 'int' <LValueToRValue>  
|         | ` -DeclRefExpr 0x7fe550043e10 <col:10> 'int' lvalue ParmVar 0x7fe550043c00 'a' 'int'  
|         ` -ImplicitCastExpr 0x7fe550043e68 <col:14> 'int' <LValueToRValue>  
|           ` -DeclRefExpr 0x7fe550043e30 <col:14> 'int' lvalue ParmVar 0x7fe550043c80 'b' 'int'
```

# Simple code example: Clang AST representation

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

```
| -FunctionDecl 0x7fe550043d60 <line:5:1, line:7:1> line:5:5 used sum 'int (int, int)'  
| | -ParmVarDecl 0x7fe550043c00 <col:9, col:13> col:13 used a 'int'  
| | -ParmVarDecl 0x7fe550043c80 <col:16, col:20> col:20 used b 'int'  
| ` -CompoundStmt 0x7fe550043eb0 <col:23, line:7:1>  
|   ` -ReturnStmt 0x7fe550043ea0 <line:6:3, col:14>  
|     ` -BinaryOperator 0x7fe550043e80 <col:10, col:14> 'int' '+'  
|       | -ImplicitCastExpr 0x7fe550043e50 <col:10> 'int' <LValueToRValue>  
|         | ` -DeclRefExpr 0x7fe550043e10 <col:10> 'int' lvalue ParmVar 0x7fe550043c00 'a' 'int'  
|       ` -ImplicitCastExpr 0x7fe550043e68 <col:14> 'int' <LValueToRValue>  
|         ` -DeclRefExpr 0x7fe550043e30 <col:14> 'int' lvalue ParmVar 0x7fe550043c80 'b' 'int'
```



# Simple code example: LLVM IR representation

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

```
; Function Attrs: noinline nounwind optnone ssp uwtable  
define i32 @_Z3sumii(i32, i32) #0 {  
    %3 = alloca i32, align 4  
    %4 = alloca i32, align 4  
    store i32 %0, i32* %3, align 4  
    store i32 %1, i32* %4, align 4  
    %5 = load i32, i32* %3, align 4  
    %6 = load i32, i32* %4, align 4  
    %7 = add nsw i32 %5, %6  
    ret i32 %7  
}
```

# Simple code example: LLVM IR representation

```
int sum(int a, int b) {  
    return a - b;  
}
```

```
; Function Attrs: noinline nounwind optnone ssp uwtable  
define i32 @_Z3sumii(i32, i32) #0 {  
    %3 = alloca i32, align 4  
    %4 = alloca i32, align 4  
    store i32 %0, i32* %3, align 4  
    store i32 %1, i32* %4, align 4  
    %5 = load i32, i32* %3, align 4  
    %6 = load i32, i32* %4, align 4  
    %7 = sub nsw i32 %5, %6  
    ret i32 %7  
}
```

# Mutation testing: Application domains

- Pure computation, algorithms
- System software, POSIX calls, IOs
- Embedded software
- "Normal" software, "business logic" ~ branch coverage

Time for a short 10-minutes break

Mull demonstration

# 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

Mull mutation testing system

# Mull, practical mutation testing tool for C and C++

- Based on LLVM
- Exists since 2016
- Built with large projects in mind
- Cross-platform (Linux, macOS, FreeBSD)
- Open Source <https://github.com/mull-project/mull>
- 10-20% project for 2 people, 19 contributors total
- Very well maintained system
  - All LLVM versions
  - Test coverage



# Mull's technical stack

- C/C++
- CMake
- LLVM 6+
- Clang AST API
- LLVM IR API
- LLVM JIT (until 2020)
- Frontend and backend parts
  - Frontend: two C/C++ frontends, Swift frontend (external project)
    - Frontend finds and records the mutations for execution
  - Backend - Mull Runner
    - Runs the mutated programs and collects the test results

# Mull's algorithm (1)

```
#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;
}
```

# Mull's algorithm (2)

```
#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;
}
```

# Mull's algorithm (3)

```
#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;
}

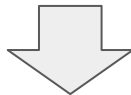
int sum_mutant_2(int a, int b) {
    return a / b;
}
```

## Mull's algorithm (4)

- 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)
  - New implementation: execute each mutant as a separate child process
- Report results

# Mull's AST-based algorithm (in works)

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



```
int sum(int a, int b)  {  
    return getenv("cxx_add_to_sub:/sandbox/mull/tests/cxx-frontend/sample.cpp:6:12") ? a - b : a + b;  
}
```



```
$ env "cxx_add_to_sub:/sandbox/mull/tests/cxx-frontend/sample.cpp:6:12 "=1  
/sandbox/mull/tests-lit/tests/cxx-frontend/00-sandbox/sample.cpp.exe
```

```
$ mull-runner /sandbox/mull/tests/cxx-frontend/sample.cpp.exe
```

# Mull's 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>

# Mull's algorithm - Sandboxing

- Clean state for each run
- Sandboxing
  - mutated code can crash
  - mutated code can hit deadlock/infinite loop
  - mutated code can sometimes crash
- `fork()` / `exec()`
  - Mutations are run as child processes
- Not a good idea to run Mull with `sudo`



# Mull's performance

- Reasonably fast
- OpenSSL (300 KLoC)
  - 2018: 188 seconds
  - 2020: 17 seconds
- LLVM (1.300 KLoC)
  - 2018: 3h 26m
  - 2020: ?

# Incremental mutation testing

- Working against mutations found in the Git changesets
  - `<mutation testing tool> --git-branch=origin/master`
- Significant reduction of number of mutants
- Easier integration into developer workflow
- Used "at scale" at Google:
  - Paper "State of Mutation Testing at Google" (2018)

# How Mull is tested

- Google Test
  - Unit tests
- LLVM LIT and FileCheck (LLVM Integrated Tester)
  - Integration tests
  - Main testing method
  - Have ported FileCheck to Python to simplify using within Mull
- Auto-testing has been tried but not integrated into the main tree
  - Remember: everyone is busy
- Continuous Integration
  - Provisioned with Ansible
  - GitHub Actions

# Problems of mutation testing

- Execution time
  - Applying Mutation Analysis On Kernel Test Suites: An Experience Report. <https://ieeexplore.ieee.org/document/7899043/>, ~3500 hours!
  - Mutation Analysis for Cyber-Physical Systems: Scalable Solutions and Results in the Space Domain, <https://arxiv.org/abs/2101.05111>, 31878 hours on an HPC cluster.
- Selection of mutants that results in good quality with less effort
  - Redundant mutants
- Human Oracle problem
  - Analysis time
- "Random" nature of mutation => rather "bottom-up" verification methods
- Equivalent mutants

# Equivalent mutants examples

```
void doSomething() {  
    // actually do something...  
    printf("doSomething() is called"); // -> 42  
}
```

```
int max(int a, int b) {  
    if (a > b) { // -> >=  
        return a;  
    } else {  
        return b;  
    }  
}
```

```
int doSomething() {  
    if (fastVersion) { // -> !fastVersion  
        // do something  
    } else {  
        // do something but slower  
    }  
    return result...  
}
```

# Mutation testing: State of the art

- Tools exist for various programming languages
  - Different levels of maturity
- Not a widespread use
  - Adoption by tool developers
  - Case studies (academia, research grants)
  - Included in portfolio by some commercial tools
- Performance, ease of integration into developer workflow
  - Too opinionated reporting
  - Two modes: full analysis and incremental analysis
- The most prominent example so far:
  - Use "at scale" at Google
    - Paper "State of Mutation Testing at Google" (2018)

# Current and future work on Mull

- AST-level mutations
- Test suite reduction
  - Redundant tests detection
- Equivalent mutants detection
- Mutation testing for embedded software
  - LLVM has several backends for embedded targets
- MC/DC coverage tool within Mull framework

# Lessons learned

- A promising "cool" technology is not necessarily the way to go
  - Constrained by the limitations of LLVM JIT
- A large slice of time is spent on maintenance
  - Fighting the CI (Travis, GitHub Actions)
  - Supporting LLVM on a number of operating systems
  - See "Building an LLVM-based tool: lessons learned",  
<https://www.youtube.com/watch?v=Yvj4G9B6pcU>
- It is not much that you can do with your free time :)
- A marathon, not a sprint project



# Summary

- What is critical software
- Overview of the testing methods
- Mutation testing method
- Mull mutation testing system
  - LLVM compiler infrastructure

# Links

- <https://mull.readthedocs.io/en/latest/GettingStarted.html>
- <https://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>

Other tools and approaches:

- <https://github.com/theofidry/awesome-mutation-testing>