Binary Mutation Analysis of Tests Using Reassembleable Disassembly

Navid Emamdoost Vaibhav Sharma Taejoon Byun Stephen McCamant

UNIVERSITY OF MINNESOTA
Driven to Discover℠

Motivation

University of Minnesota

- Software usage is increasingly common in embedded/critical systems
  - security cameras, industrial vacuum cleaners, nuclear power plants
  - 3rd party components are prevalent
- System-level integrators often rely on 3rd party binary-only libraries
- Security properties like exploitability can only be determined on final binary

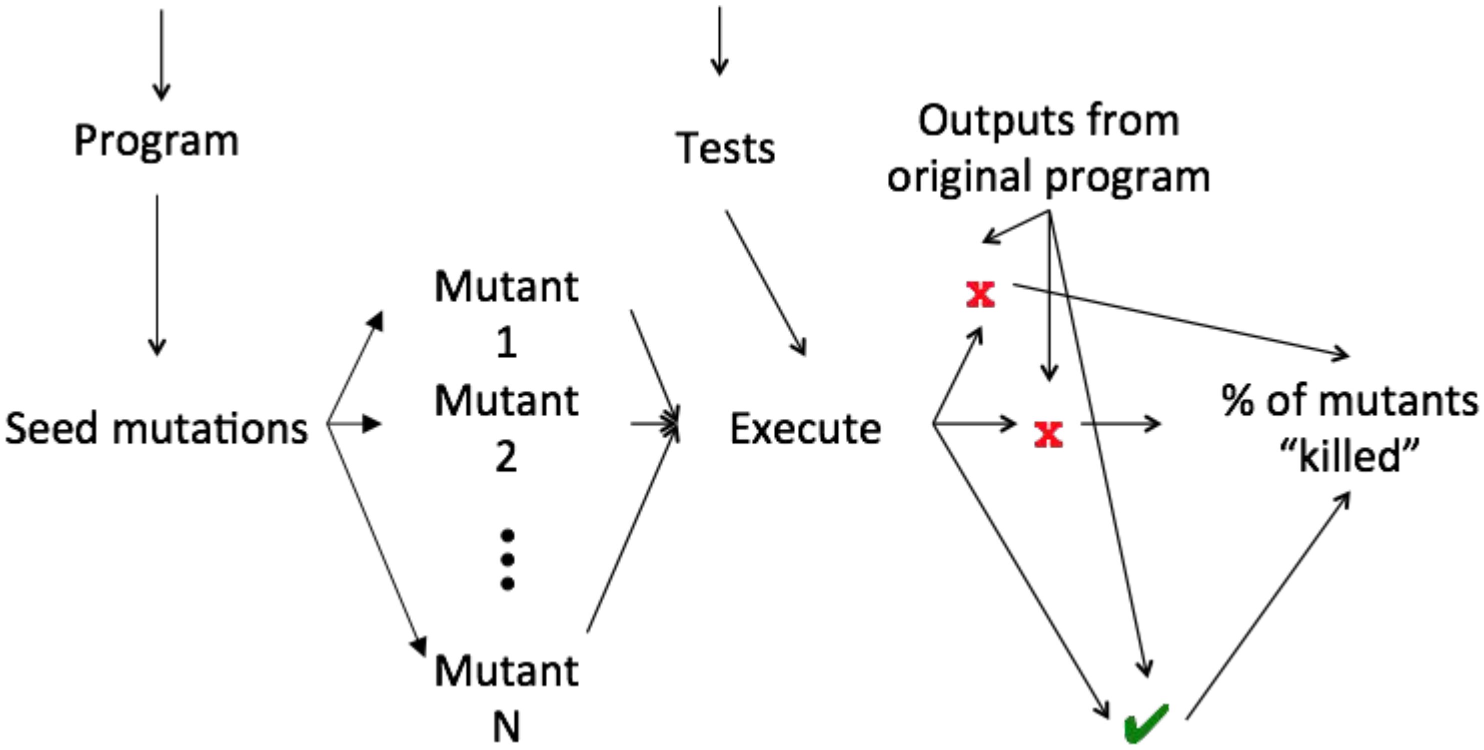
Motivation

UNIVERSITY OF MINNESOTA

- Software testing is important!
- Regression tests are based on bugs found in the past
- How can we know how “good” our tests are?
- Mutation introduces a small change in the program
    - Used as a proxy for real bugs
    - Need not be hard-to-find bugs
    - Want all mutations to be detected, regardless of what behavior they introduce

How does Mutation Analysis work?

UNIVERSITY OF MINNESOTA



Mutation Analysis For Binaries

University of Minnesota

- What instructions should we mutate?
  - conditional jump instructions (jCC)
  - conditional move instructions (cmovCC)
  - set byte on condition instructions (setCC)
  - add/subtract with carry/borrow (adc/sbb)

How do these mutation operators correspond to bugs?

Mutation Example (jCC)

```
static int duplicate_decls (
    tree newdecl, tree olddecl, int different_binding_level)
{
    ...
    /* begin added code */
    else if (TYPE_ARG_TYPES (oldtype) == NULL
                && TYPE_ARG_TYPES (newtype) != NULL) {
        ...
    } /* end added code */
    ...
}
```

Mutation Example (jCC)

UNIVERSITY OF MINNESOTA

```
. . .
805c9c5: mov  0x18(%esp),%eax
805c9c9: mov  0xc(%eax),%ecx
805c9cc: test %ecx,%ecx
; TYPE_ARG_TYPES (oldtype) == NULL
805c9ce: jne  805bda4 <duplicate_decls+0x134>
805c9d4: mov  0xc(%edi),%eax
805c9d7: test %eax,%eax
; TYPE_ARG_TYPES (newtype) != NULL
805c9d9: je   805bda4 <duplicate_decls+0x134>
. . .
```

Mutation Example (jCC)

Mutating this jne to a unconditional jump reverts the patch

```
. . .
805c9c5: mov  0x18(%esp),%eax
805c9c9: mov  0xc(%eax),%ecx
805c9cc: test %ecx,%ecx
; TYPE_ARG_TYPES (oldtype) == NULL
805c9ce: jne  805bda4 <duplicate_decls+0x134>
805c9d4: mov  0xc(%edi),%eax
805c9d7: test %eax,%eax
; TYPE_ARG_TYPES (newtype) != NULL
805c9d9: je   805bda4 <duplicate_decls+0x134>
. . .
```

How do we construct binary mutants?

On Binary Rewriting

UNIVERSITY OF MINNESOTA

- In place instruction rewriting
  - Challenging when the new instruction is longer
- Dynamic instruction rewriting
  - Test suite runtime overhead
- Reassembleable Disassembly
  - Reusable tool translates binary into assembly
  - Make changes statically to the assembly
  - Reassemble to binary
  - Run test suite without runtime overhead

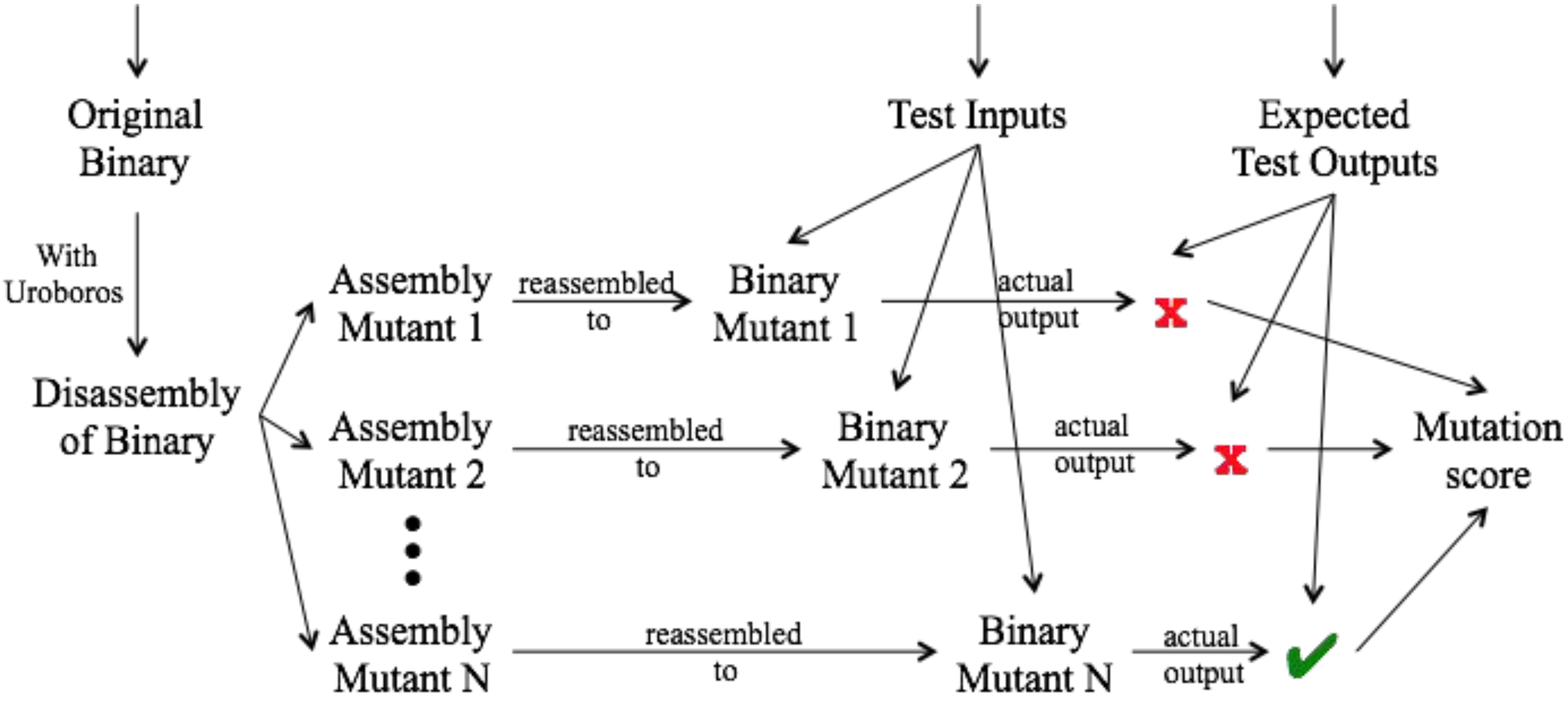
Reassembleable Disassembly

UNIVERSITY OF MINNESOTA

- Two available open-source projects:
  - Uroboros [USENIX'15]
    - `https://github.com/s3team/uroboros`
  - Ramblr [NDSS'17]
    - `https://github.com/angr/patcherex`
- Uroboros worked fine on most of our binaries
  - Some reliability issues on 2 larger binaries
- Gave Ramblr a shot
  - No success on non-CGC binaries

Binary Mutation Analysis Workflow

University of Minnesota



Mutant Categories

- Mutants differ from the original binary only in mutated instruction
- Based on the result of mutant execution
  - **Killed**: mutants that does not produce test's expected output
  - **Live**: mutants that produced the expected test's output
  - **Trivial**: mutants that fail on any input (excluded from killed)

Measuring the Test Quality

University of Minnesota

- Mutation Score
  - #(killed mutants) / #(total mutants)
- Mutant coverage
  - Killed mutants are covered by test input
  - Live mutants may or may not be covered
    - Input may reach the mutated instruction
      - But is not reflected in the output
    - An example of one such mutant is provided in the paper

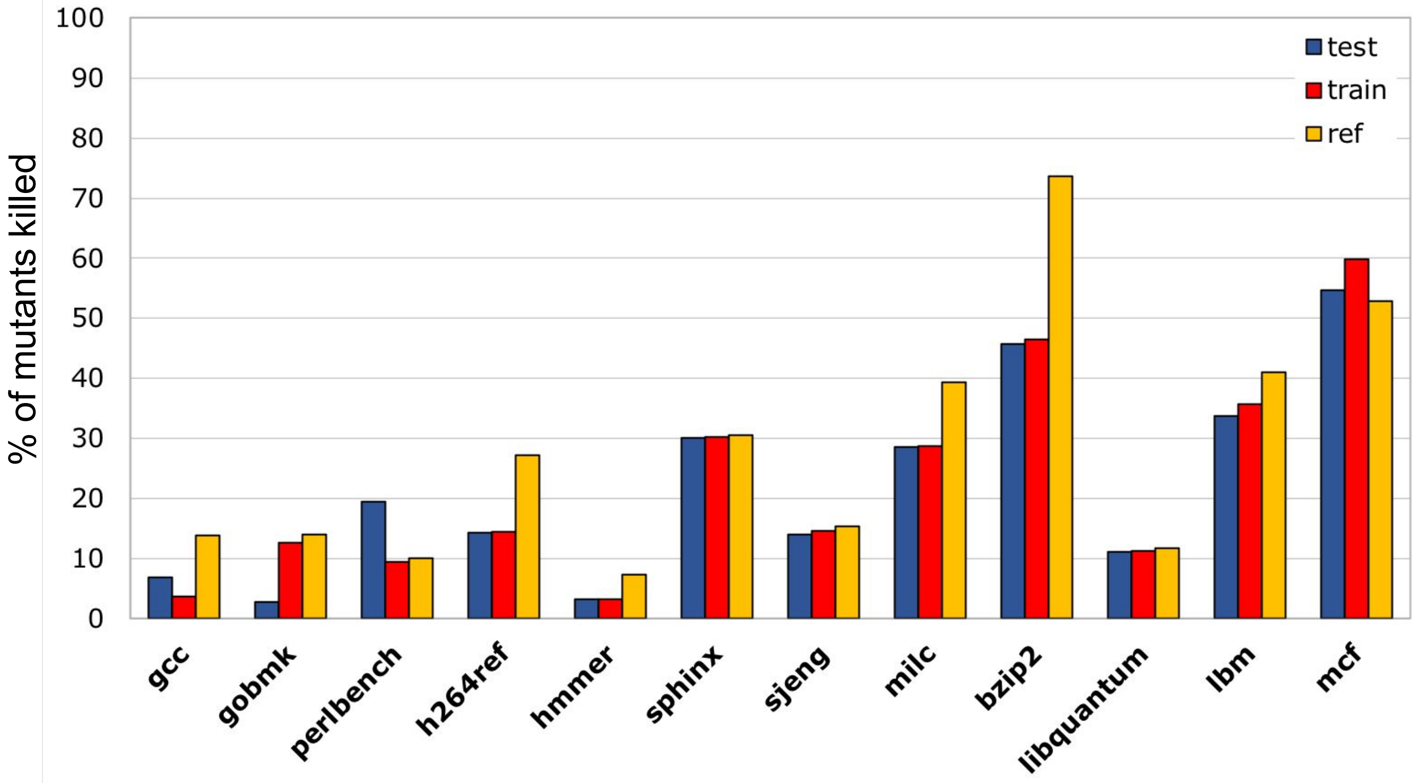
UNIVERSITY OF MINNESOTA

Evaluation

SPEC 2006

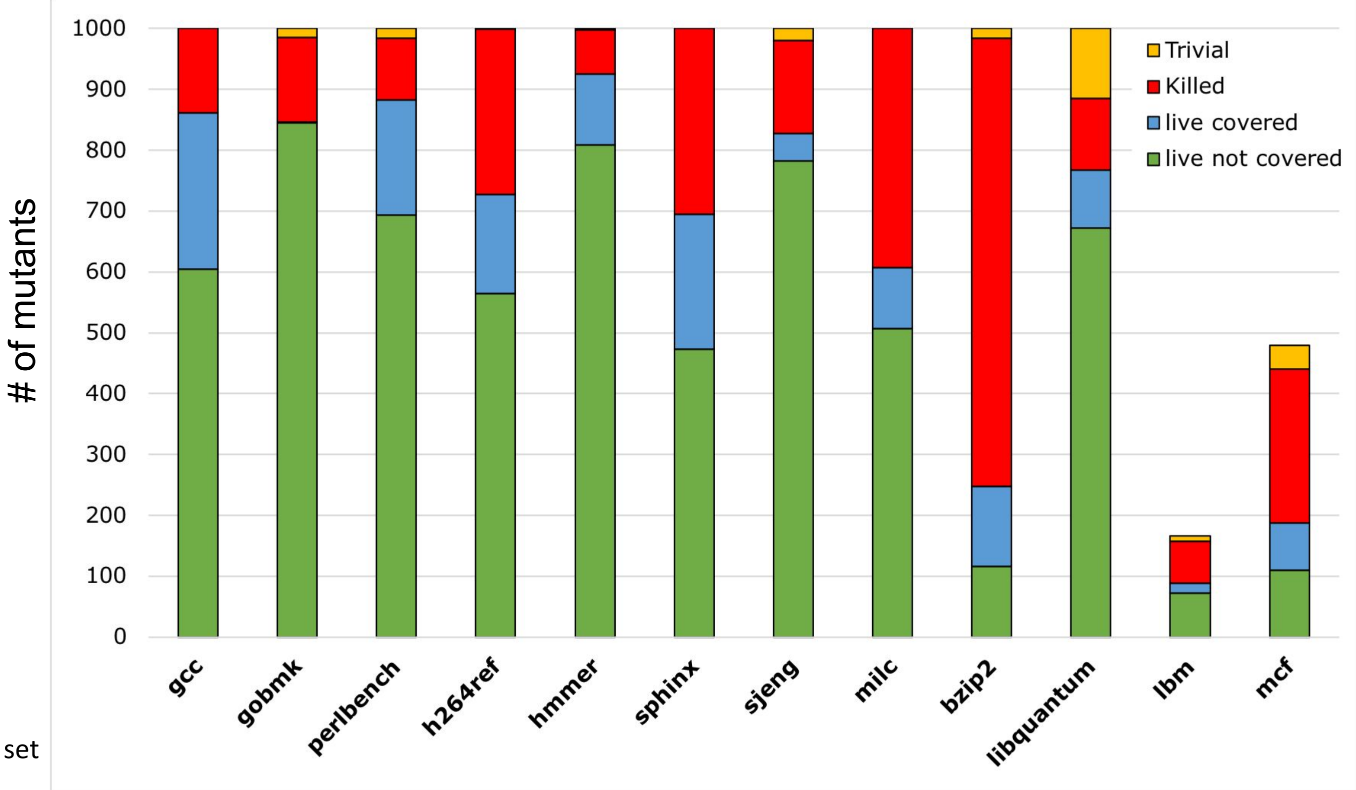
- 12 benchmarks in C
- 3 different input sets
  - **test**: to confirm the binary is functional
  - **train**: used for feedback-driven optimization
  - **ref**: the actual workload
- Generated as many mutants as possible
  - Select 1000 randomly for each binary
  - For *lbm* and *mcf*: 166 and 480 mutants respectively

SPEC 2006 - Mutation Score



SPEC 2006 - Categorized Mutants*

University of Minnesota



*on **ref** input set

Embedded-Control Binaries

UNIVERSITY OF MINNESOTA

- Selected 5 safety-critical binaries
  - Docking approach application
  - Cruise controller
  - Infusion pump
  - Microwave control logic (two versions)
- The test input generated automatically using a coverage-guided test generation technique [1]

1. T. Byun, V. Sharma, S. Rayadurgam, S. McCamant, and M. P. E. Heimdahl, “Toward rigorous object-code coverage criteria,” in 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), Oct 2017, pp. 328–338.

Mutation Results

University of Minnesota

Embedded Binary	Trivial Mutants	Killed Mutants	Live Mutants	Binary Mutation Score	Source-level Mutation Score
Docking Approach	0	1154	1854	38.4%	26.9%
Infusion Pump	0	525	723	42.1%	25.0%
Cruise Controller	25	594	277	68.1%	73.6%
Microwave (auto)	22	390	150	72.2%	67.8%
Microwave (manual)	1	86	11	88.6%	67.8%

Future Work

University of Minnesota

- Improve the variety of mutation operators
  - Make binary mutants more representative of the source-level mutants
- Explore practicality of in-place binary rewriting for mutant generation
- Perform a comprehensive comparison between source-level and binary-level mutation

Q/A

University of Minnesota

Thank you for your attention

Check out:

`https://github.com/navidem/binarymutation`