SN4KE: Practical Mutation Testing at Binary Level

BAR@NDSS2021

Mohsen Ahmadi **Pantea Kiaei** **Navid Emamdoost**

ARIZONA STATE UNIVERSITY

WPI

UNIVERSITY OF MINNESOTA
Driven to Discover℠

Motivation

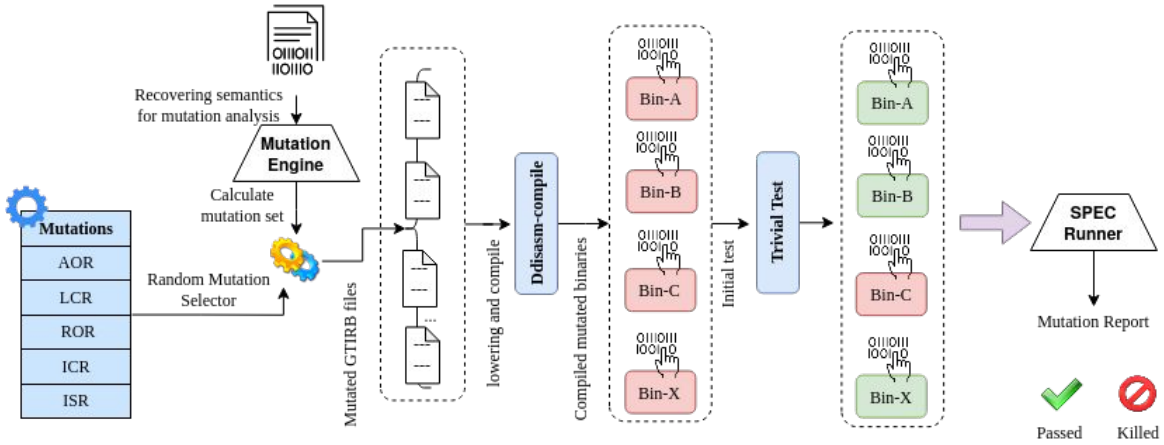
- Software/firmware usage is increasingly common in embedded/critical systems
  - security cameras, industrial vacuum cleaners, nuclear power plants
  - 3rd party components are prevalent
    - Bluetooth, Wi-fi
    - Encryption libraries (i.e wolfcrypt)
    - Manufacturer components (i.e. Broadcom, Qualcomm, Sierra, etc)

- System-level integrators often rely on 3rd party binary-only libraries
- Security properties like exploitability can only be determined on final binary

Motivation

- 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 Binary Mutation Testing work?



Binary Mutation Testing

Mutation class	Description	Example
Arithmetic	1. Replace arithmetic assignment operator from the set {+=, -=, *=, /=} 2. Replace with an operator from the set of {+, -, *, /, %}	MOV ECX, count REP **DEC** DWORD PTR ES:[ESI] ->\ MOV ECX, count REP **INC** DWORD PTR ES:[ESI]
Logical	1. Substitute with another bitwise logical operator from {^, |, &} 2. Replace with a logical assignment from {^=, |=, &=} 3. Substitute connector with another logical operator from {&&, ||}	**XOR** EDI, EDI INC EDI CMP EDI, \xFF ->\ **OR** EDI, EDI INC EDI CMP EDI, \xFF
Conditional	Substitute any conditional jump with an unconditional branch or NOPing the condition	%cmp = icmp **slt** i32 %2, 10 br i1 %cmp, label %if.then, label %if.else`. ->\ %cmp = icmp **ne** i32 %2, 10 br i1 %cmp, label %if.then, label %if.else`.

Constants	Replace any immediate value *c* with one another constant from set {-1, 0, 1, -c, c+1, c-1}	add r0, r0, %1
		->\
		add r0, r0, %0

Skip	Skip executing an instruction by replacing any of Arithmetic, Logical or Conditional classes with NOP instruction	**XOR** EDI, EDI INC EDI CMP EDI, \xFF ->\ **NOP; NOP** INC EDI CMP EDI, \xFF

How do these mutation operators represent bugs?

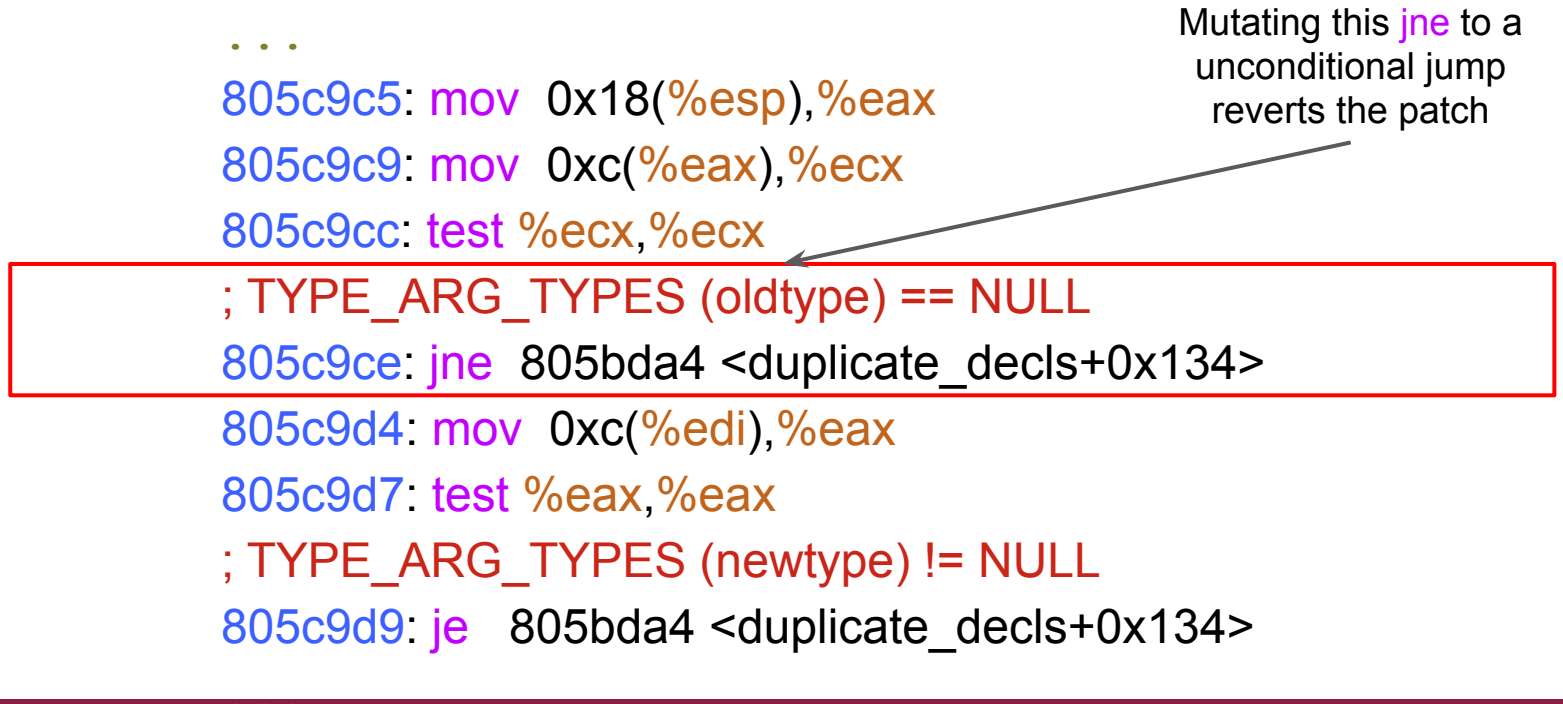
Mutation Example (GCC)

```
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 (GCC)

```
. . .
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 (GCC)



Compiler undefined behavior

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
    return; /* len too large */
if (buf + len < buf)
    return; /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

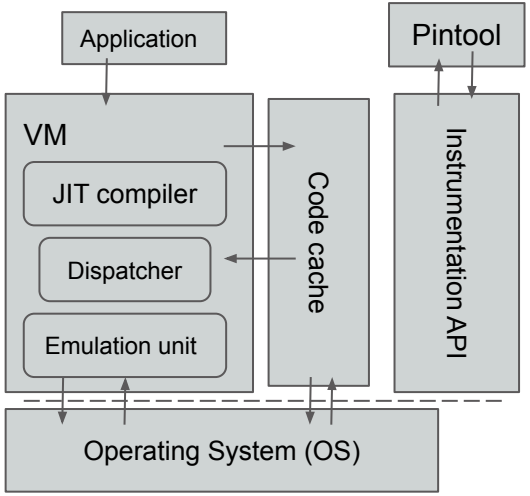
A pointer overflow check removed during compilation optimization using GCC because it optimize away the second *if* statement silently.

*CMU CERT: https://www.kb.cert.org/vuls/id/162289

How do we construct binary mutants?

On Binary Rewriting

- Dynamic
  - PIN, DynamoRIO
  - QEMU, Valgrind
- Static
  - Detouring: hooks the underlying instruction
    - Patch-based
    - Replica-based
  - Reassembleable disassembly: recovering .reloc table
    - UROBOROS, Ramblr, ddisasm (Datalog disassembly)
  - Full-translation
    - Rev.ng



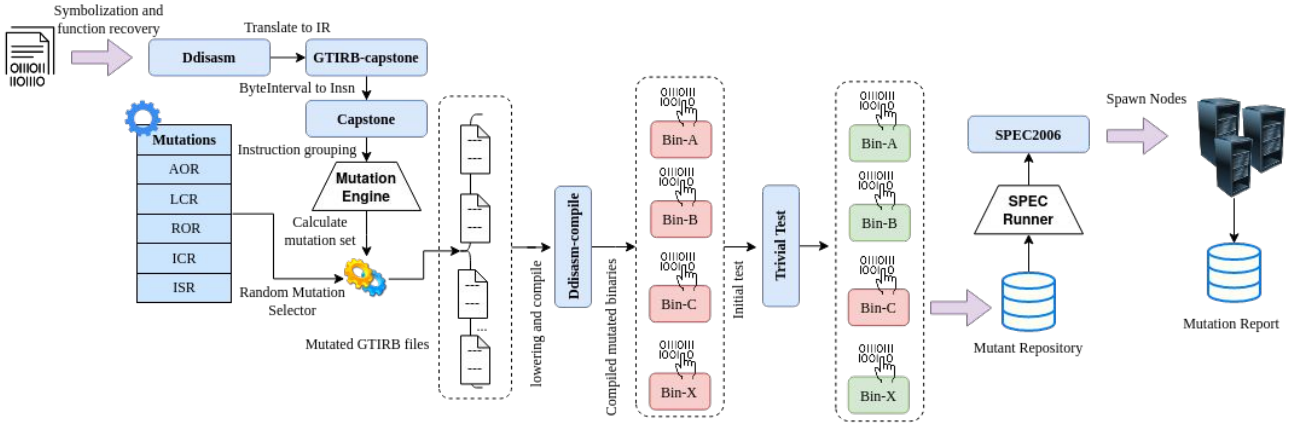
Reassembleable Disassembly vs. Full-translation

- Two available open-source projects:
  - Rev.ng [ICCST'18]
    - https://github.com/revng
  - ddisasm [USENIX'20]
    - https://github.com/GrammaTech/ddisasm

- ddisasm worked perfect on all the binaries in SPEC
  - UROBOROS reported problems with two binaries
  - Also, dependant on a specific compiler version

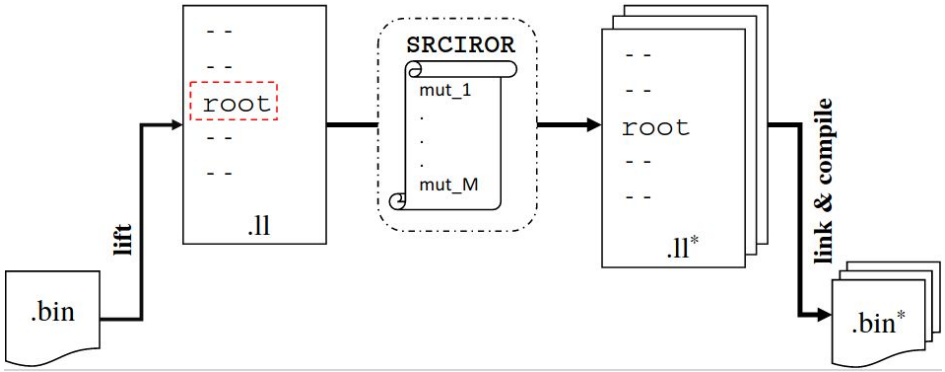
- Rev.ng oversized the final results by x35-x75 times

SN4KE Workflow: ddisasm



SN4KE workflow consists of four stages. First, we pass the binary under test to ddisasm for relocation table reconstruction and performing symbolization on binary. The resulting GTIRB file is then passed to the mutation engine, where we randomly apply a chosen technique. Next, we use ddisasm-pprinter to reassemble the transformed GTIRB into an executable. To make sure the binary is passing the initial checks, we ran it through the trivial test. Successful candidates are then passed to the SPEC runner to get the mutation report.

SN4KE Workflow: rev.ng



*SRCIROR: https://github.com/TestingResearchIllinois/srciror

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

- Mutation Score
  - #(killed mutants) / #(total mutants)
- Mutant coverage
  - Killed mutants are covered by test input
  - Passed 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

Evaluation

SPEC 2006

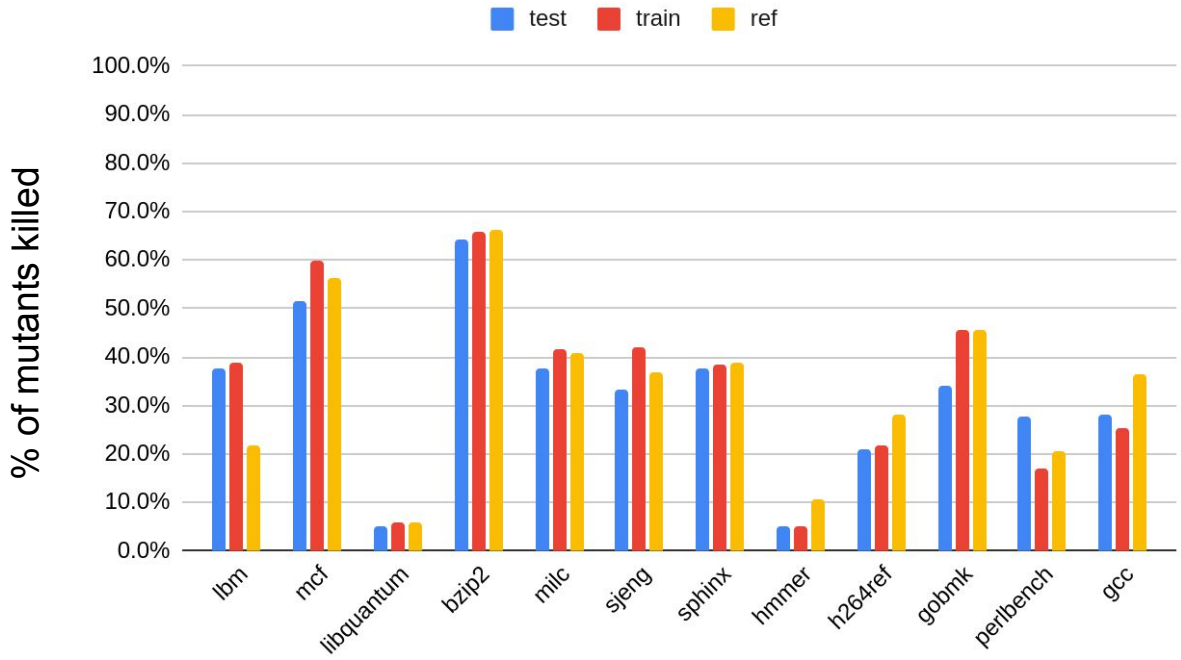
- 12 benchmarks in C
- 3 different input sets
  - **test**: to confirm the binary is functional
  - **train**: used for feedback-driven optimization
  - **ref**: the actual workload

- Generated as many mutants as possible
  - Select 1000 randomly for each binary
  - For *lbm* we could only generate 641 mutants

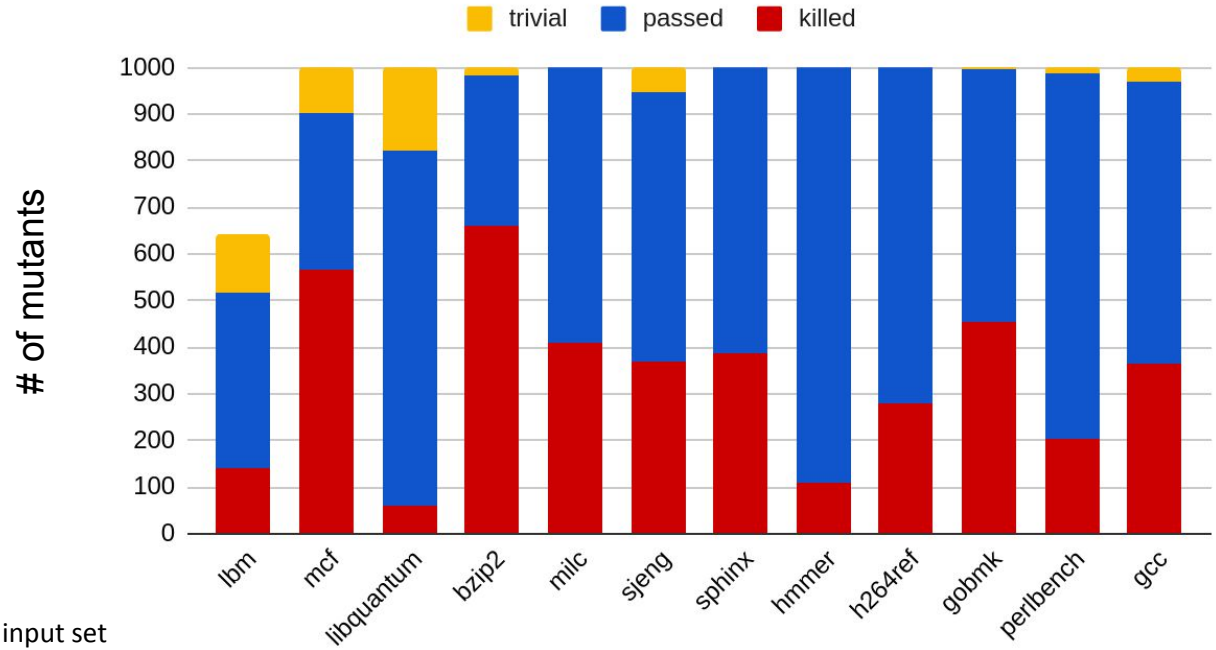
Number of possible mutations

	ddisasm		rev.ng	
	GTIRB size	# mutations	LLVM IR size	# mutations
lbm	190 KB	641	14 MB	53215
mcf	322 KB	1311	14 MB	47390
libquantum	898 KB	21277	24 MB	177310
bzip2	1.5 MB	32426	30 MB	270217
milc	2.6 MB	57199	46 MB	476663
sjeng	3.4 MB	59572	53 MB	496435
sphinx	4.4 MB	83918	63 MB	699322
hmmer	7.6 MB	147460	103 MB	1228840
h264ref	12 MB	253270	177 MB	2110590
gobmk	43 MB	571174	367 MB	4759791
perlbench	97 MB	1574122	986 MB	13117690
gcc	28 MB	420863	283 MB	3507198

SPEC 2006 - Mutation Score

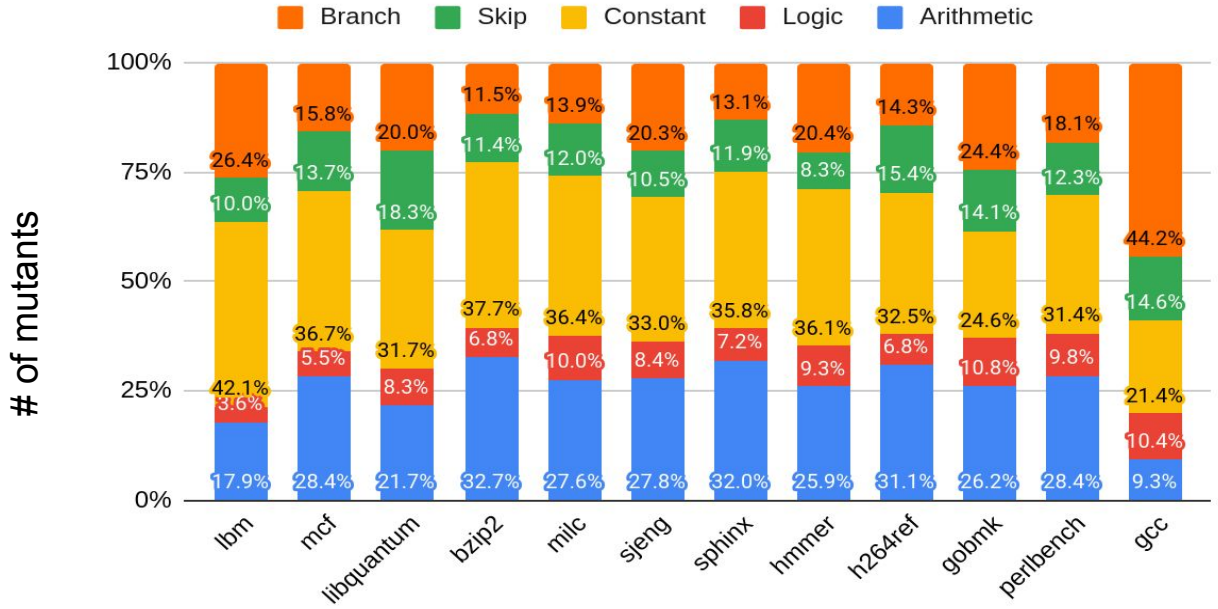


SPEC 2006 - Categorized Mutants*



*on **ref** input set

SPEC 2006 - Categorized Mutants*

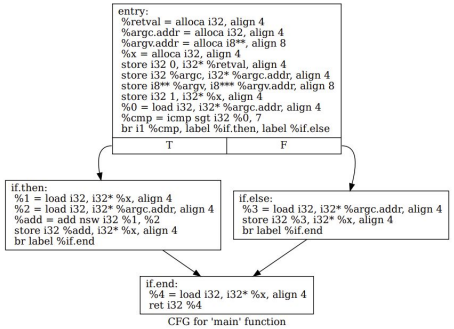


*on **ref** input set

Rev.ng limitations - Binary size

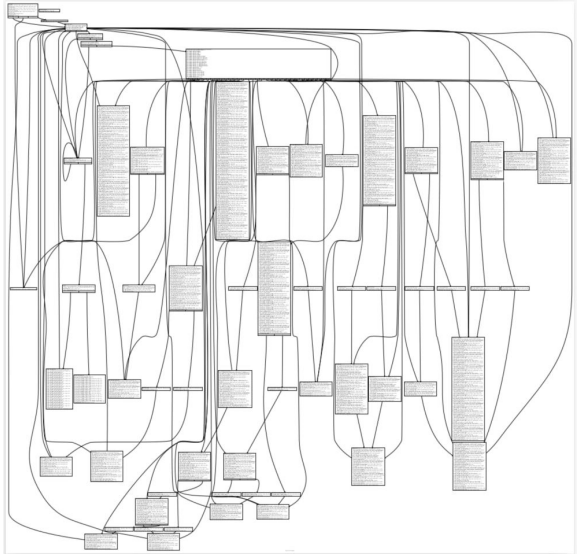
```
int main(int argc, char* argv[]) {

  int x = 1;
  if (argc > 7 )
    x = x + argc;
  else
    x = argc;

  return x;
}
```

source-code



Original CFG

benchmark	original size	Revng		Ddisasm	
		rewritten size	rewrite overhead	rewritten size	rewrite overhead
lbm	22kB	1.2MB	54.5×	22kB	1×
mcf	23kB	1.2MB	52.2×	22kB	0.96×
libquantum	51kB	3.5MB	68.6×	46kB	0.9×
bzip2	69kB	4,1MB	59.4×	68kB	0.99×
milc	142kB	5.7MB	40.1×	134kB	0.94×
sjeng	154kB	8.1MB	52.6×	149kB	0.97×
sphinx	198kB	7.5MB	37.9×	196kB	0.99×
hmmer	314kB	13MB	41.4×	308kB	0.98×
h264ref	566kB	22MB	38.9×	552kB	0.98×
perlbench	1.2MB	47MB	39.2×	1.5MB	1.25×
gcc	3.6MB	127MB	35.3×	3.5MB	0.97×
gobmk	3.9MB	39MB	10.0×	4.3MB	1.1×



CFG after re-compilation using rev.ng

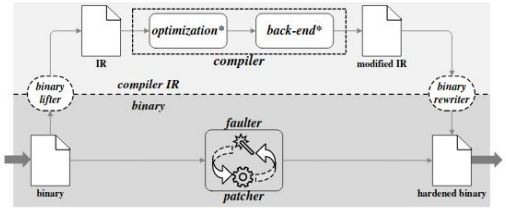
Rev.ng limitations - Runtime overhead

- Execution time slow down
  - Uroboros/ddisasm imposes 1% slow-down in reassembled binary
  - Rev.ng introduces up to 10% slow-down in re-compiled binary

- Instrumentation overhead
  - Ddisasm
    - ddisasm takes 40s on average to generate GTIRB file for a SPEC binary
    - We spend roughly 25s reassembly time for each mutant
  - Rev.ng
    - Takes 15 minutes on average to generate .llvm file
    - And roughly 4 minutes to compile it back to binary

Future Work

- Mutant selection mechanism
  - Numerous mutants can be generated, skip less interesting ones without executing
- Generating targeted mutants
  - Targeting a specific execution using Data flow information
- Making mutants more representative of real bugs
- Fault-injection countermeasures: https://arxiv.org/abs/2011.14067



Q/A

Mohsen Ahmadi
pwnslinger@asu.edu

Check out:

https://github.com/pwnslinger/sn4ke

Special Thanks to my collaborators and BAR committee

A Size-Changing Mutator

- `adc src, dst`
  - Mutant1: `add src, dst; inc dst`
  - Mutant2: `add src, dst`
- `sbb src, dst`
  - Mutant1: `sub src, dst; dec dst`
  - Mutant2: `sub src, dst`