SN4KE: Practical Mutation Testing at Binary Level

BAR@NDSS2021

Mohsen Ahmadi  Pantea Kiaei  Navid Emamdoost

Arizona State University  Worcester Polytechnic Institute  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

<table>
<thead>
<tr>
<th>Mutation class</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| **Arithmetic** | 1. Replace arithmetic assignment operator from the set \{+=, -=, *=, /=\}  
2. Replace with an operator from the set \{+, -, *, /, \%\} | MOV ECX, count  
REP DEC DWORD PTR ES:[ESI]  
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, $FF  
OR EDI, EDI  
INC EDI  
CMP EDI, $FF |
| **Conditional** | Substitute any conditional jump with an unconditional branch or NOPing the condition | %cmp = icmp $l %2, 10  
br if %cmp, label "if.then", label "if.else".  
%cmp = icmp ne $l %2, 10  
br if %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, $FF  
NOP; NOP  
INC EDI  
CMP EDI, $FF |
How do these mutation operators represent bugs?
Mutation Example (GCC)

```c
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>
...  
```
Mutating this `jne` to an unconditional jump reverts the patch.

```assembly
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>
```
Compiler undefined behavior

```c
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](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
Reassemblable Disassembly vs. Full-translation

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

Mutant Categories

- Mutants differ from the original binary only in mutated instruction
- Based on the result of mutant execution
  - **Killed**: mutants that do 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
  - \(#(\text{killed mutants}) / #(\text{total mutants})\)

- 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

<table>
<thead>
<tr>
<th></th>
<th>ddisasm GTIRB size</th>
<th>ddisasm # mutations</th>
<th>rev.ng LLVM IR size</th>
<th>rev.ng # mutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibm</td>
<td>190 KB</td>
<td>641</td>
<td>14 MB</td>
<td>53215</td>
</tr>
<tr>
<td>mcf</td>
<td>322 KB</td>
<td>1311</td>
<td>14 MB</td>
<td>47390</td>
</tr>
<tr>
<td>libquantum</td>
<td>898 KB</td>
<td>21277</td>
<td>24 MB</td>
<td>177310</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.5 MB</td>
<td>32426</td>
<td>30 MB</td>
<td>270217</td>
</tr>
<tr>
<td>milc</td>
<td>2.6 MB</td>
<td>57199</td>
<td>46 MB</td>
<td>476663</td>
</tr>
<tr>
<td>sjeng</td>
<td>3.4 MB</td>
<td>59572</td>
<td>53 MB</td>
<td>496435</td>
</tr>
<tr>
<td>sphinx</td>
<td>4.4 MB</td>
<td>83018</td>
<td>63 MB</td>
<td>696322</td>
</tr>
<tr>
<td>hmmmer</td>
<td>7.6 MB</td>
<td>147460</td>
<td>103 MB</td>
<td>1228840</td>
</tr>
<tr>
<td>h264ref</td>
<td>12 MB</td>
<td>253270</td>
<td>177 MB</td>
<td>2110590</td>
</tr>
<tr>
<td>gobmk</td>
<td>43 MB</td>
<td>571174</td>
<td>367 MB</td>
<td>4759791</td>
</tr>
<tr>
<td>perlbench</td>
<td>97 MB</td>
<td>1574122</td>
<td>996 MB</td>
<td>13117690</td>
</tr>
<tr>
<td>gcc</td>
<td>28 MB</td>
<td>420863</td>
<td>283 MB</td>
<td>3507198</td>
</tr>
</tbody>
</table>
SPEC 2006 - Mutation Score

% of mutants killed

- test
- train
- ref

ibm, ncf, ilquantum, bzip2, milc, sjeng, sphinx, hammer, h264ref, gobmk, perlbench, gcc
SPEC 2006 - Categorized Mutants*

*on ref input set

# of mutants

- trivial
- passed
- killed

- ibm
- mcf
- libquantum
- bzip2
- milc
- sjeng
- sphinx
- hmmer
- h264ref
- gobmk
- perlbench
- gcc
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

```
<table>
<thead>
<tr>
<th>benchmark</th>
<th>original size</th>
<th>revng rewritten size</th>
<th>revng rewrite overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>libquantum</td>
<td>51kB</td>
<td>3.5MB</td>
<td>52.2 ×</td>
</tr>
<tr>
<td>bzip2</td>
<td>69kB</td>
<td>4.1MB</td>
<td>59.4 ×</td>
</tr>
<tr>
<td>mile</td>
<td>142kB</td>
<td>5.7MB</td>
<td>40.1 ×</td>
</tr>
<tr>
<td>gjpeg</td>
<td>134kB</td>
<td>8.1MB</td>
<td>56.2 ×</td>
</tr>
<tr>
<td>sphinxx</td>
<td>198kB</td>
<td>7.5MB</td>
<td>37.9 ×</td>
</tr>
<tr>
<td>hmmer</td>
<td>334kB</td>
<td>13MB</td>
<td>41.4 ×</td>
</tr>
<tr>
<td>l2stopref</td>
<td>566kB</td>
<td>22MB</td>
<td>38.9 ×</td>
</tr>
<tr>
<td>perlbench</td>
<td>1.2MB</td>
<td>47MB</td>
<td>39.2 ×</td>
</tr>
<tr>
<td>gcc</td>
<td>3.6MB</td>
<td>127MB</td>
<td>35.3 ×</td>
</tr>
<tr>
<td>pobox</td>
<td>3.9MB</td>
<td>39MB</td>
<td>10.0 ×</td>
</tr>
</tbody>
</table>
```

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
Q/A

Thank you

Check out:
https://github.com/pwnslinger/sn4ke

Special Thanks to my collaborators and BAR committee
A Size-Changing Mutator

- \texttt{adc src, dst}
  - \textbf{Mutant1:} \texttt{add src, dst; inc dst}
  - \textbf{Mutant2:} \texttt{add src, dst}

- \texttt{sbb src, dst}
  - \textbf{Mutant1:} \texttt{sub src, dst; dec dst}
  - \textbf{Mutant2:} \texttt{sub src, dst}