Micro Execution

Patrice Godefroid
Microsoft Research
What is Micro Execution?

Micro Execution is the ability to run any code fragment without a user-provided test driver or input data.
What is Micro Execution?

Micro Execution is the ability to run any code fragment without a user-provided test driver or input data.

VM for test isolation and generation

```c
void foo(char *p) {  // p is a 4-byte input
    char v = *p;     // *p is a 1-byte input
    return;
}
```

can execute any code
intercepts all memory operations
allocates memory
provides input values
What is Micro Execution?

Micro Execution is the ability to run any code fragment without a user-provided test driver or input data:

- The user selects any function or code location in any dll/exe
- A runtime VM starts executing the code at that location, catches all memory operations before they occur, and provides input values according to a customizable memory policy.

   Ex: “an input is any value read from an uninitialized function argument, plus any dereference to a previous input (recursive definition)”

```c
void foo(char *p) {  // p is a 4-byte input
  char v = *p;      // *p is a 1-byte input
  return;
}
```

Note: under this policy, uninitialized global-var reads are not inputs (other memory policies can be defined)
**MicroX**

- **MicroX** is a first prototype VM allowing micro execution of x86 binary code
  - Implemented as an extension of Nirvana (processor emulator)
  - Execute any x86 code in any (user-mode) Windows dll or exe
  - No source code, no pdb required
  - The user defines the starting point
  - Use a default memory policy, or define a new one...
  - Input values can be generated randomly, be zero, read from a file, read from a process dump, or be generated by **SAGE**
    - SAGE = tool for dynamic test generation with SMT constraint solving, widely used at Microsoft for security testing (see [ICSE'2013])
  - Stops when crash, max instr count reached, exec leaves the dll,...
  - No test driver required:
    - Inputs/Outputs are discovered dynamically by **MicroX**
Example

void foo(char *p) {  // p is a 4-byte input
    char v = *p;      // *p is a 1-byte input
    return;
}

is compiled into (x86)

1:   push ebp                ; foo starts here
2:   mov  ebp, esp
3:   push ecx
4:   mov  eax, DWORD PTR [ebp+8] ; p
5:   mov  cl, BYTE PTR [eax]   ; *p
6:   mov  BYTE PTR [ebp-1], cl ; v
7:   mov  esp, ebp
8:   pop  ebp
9:   ret   0

[...]  micro executed

1:   initEIP is 72B51005
2:   initEBP is 001EF988
3:   Read Mem Access at address 001EF990 of 4 bytes
4:   Initializing 4 input bytes:
6:   Adding 00201478 to list of known addresses
7:   SetGuestEffectiveAddress returned 00201440
8:   Read Mem Access at address 00201478 of 1 bytes
9:   Initializing 1 input bytes: [0]=29
10:  SetGuestEffectiveAddress returned 0020C490
11: Write Mem Access at address 001EF987 of 1 bytes
12:  SetGuestEffectiveAddress returned 001EF987
13: END: ExitProcess is called
14: ***** External Memory Stats: *****
15: Number of Mem Accesses: 2 (2 Reads, 0 Writes)
16: Number of Addresses: 2 (total 5 bytes)
17: Number of Inputs: 2 (total 5 bytes)
18: ***** Native Memory Stats: *****
19: Number of Module Accesses: 0 (0 Reads, 0 Writes)
20: Number of Other Accesses: 1 (0 Reads, 1 Writes)
21: ***** General Stats: *****
22: Number of Unique Instructions After Start: 9
23: Number of Warnings: 0
24: Number of Errors: 0
How is MicroX implemented?

• Program instrumentation: using micro-operations (not new)

  mov eax, [ecx]
  ...
  GenerateEffectiveAddress
  ...
  PREMemoryAccessCallBack
  ...
  mov eax, [EffectiveAddress]
  ...

  micro-operations

  New: EffectiveAddress can be hijacked here

• External memory manager
  - Maps program-visible addrs to invisible ExternalMemory addrs
  - Maintains R/W consistency, consistent addressing (ptr arith,...)
  - 100% dynamic, see paper for details

• Input value generation
  - Random, zero, native, file, process-dump modes
  - Next iterations can be generated with SAGE
Limitations of Micro Execution

- **False Positives (spurious bugs)**
  - Micro execution makes sense mostly if all inputs are unconstrained
  - Otherwise, crashes may be unrealistic, and guidance is needed to specify realistic input constraints, either by the user or by a whole program analysis tool (SAGE...)

- **False Negatives (missed bugs)**
  - May miss bugs if input set is too small (e.g., ignore a global variable)
    → adjust memory policy
  - Poor test coverage? Use dynamic test generation (SAGE), ...

- **Can only find bugs that are local to the code under test**

The next applications largely avoid those limitations
  - Work in progress
Application 1: API Fuzzing

- New API fuzzer packaging MicroX+SAGE:
  - Specify a dll name and a list of dll-exported functions
    - No need for number of args, types, test driver!
  - Automatically run MicroX+SAGE on each function for 1min

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Unique Instructions (avg [min-max])</th>
<th>Inputs (avg [min-max])</th>
<th>Memory Accesses (avg [min-max])</th>
<th>Tests</th>
<th>Crashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>_snwscanf_s</td>
<td>164 [76-388]</td>
<td>5 [1-7]</td>
<td>60 [23-155]</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>_splitpath_s</td>
<td>142 [142-142]</td>
<td>89 [37-221]</td>
<td>431 [170-1090]</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>_strnset_s</td>
<td>82 [48-139]</td>
<td>74 [3-215]</td>
<td>201 [8-636]</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>_strset_s</td>
<td>81 [30-128]</td>
<td>27 [1-253]</td>
<td>105 [4-754]</td>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>_ui64toa_s</td>
<td>165 [121-208]</td>
<td>5 [5-5]</td>
<td>242 [68-753]</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>_ui64tow_s</td>
<td>169 [121-209]</td>
<td>5 [5-5]</td>
<td>258 [68-1105]</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>_vsnwprintf_s</td>
<td>144 [139-153]</td>
<td>90 [7-130]</td>
<td>2172 [50-3189]</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1: Sample experimental results with 13 exported functions part of ntdll.dll.
Application 1: API Fuzzing & Diffing

- Repeat on another dll version and diff the results
  - ~1,800 dlls in c:\windows\system32 alone

- Remarks: Micro execution is...
  - Fast and automatic, zero-cost test-setup
  - Good code coverage (thanks to SAGE)
  - Generates tons of data... (ex: useful for API diffing)
Application 2: Parser Isolation & Fuzzing

• Identify parsing code buried anywhere
  - Ex: packet parsers

• Start micro executing that code
  - MicroX discovers automatically its I/O
  - Input values are initialized from a dump
  - Packet values are fuzzed with SAGE

• Note: MicroX + dump = “micro-fork”
  - State is recreated partially (no bottom stack) and lazily (on-demand)
Application 3: Targeted Fuzzing

• Fast precise analysis of components of large parsers
  - Ex: with SAGE
    • a single symbolic execution of MS Excel takes ~1 hour
      - 47Kbytes input file, ~1.5 billion x86 instructions, ~25,000 constraints
    • a single symbolic execution of one function buried in Excel, running with MicroX, may take only ~1 second!

• Automatic program decomposition
  - Identify sub-parser and fuzz them in isolation

• Compositional testing
  - Memorize the sub-parser results with symbolic test summaries
Application 4: Unit/Program Verification

- The ANI Windows parser
  350+ fcts in 5 DLLs, parsing in ~110 fcts in 2 DLLs, core = 47 fcts in user32.dll →
- Is “attacker memory safe”
  = no attacker-controllable buffer overflow
- How? Compositional exhaustive testing
  - “perfect” symbolic execution in SAGE (max precision, no divergences, no x86 incompleteness, no Z3 timeouts, etc.),
  - manual bounding of input-dependent loops (only ~10 input bytes + file size), and
  - 5 user-guided simple summaries
- And modulo fixing a few bugs... 😊
- 100% dynamic (=zero static analysis)
- 1st Windows image parser proved attacker memory safe
- See “Proving Memory Safety of the ANI Windows Image Parser using Compositional Exhaustive Testing”, MSR-TR-2013-120, with intern Maria Christakis
Application 5: Malware Detection

• Think of MicroX as an “eval(x86-code)” function
  - Can run any code to see if it uncloaks itself and then does something malicious

• Note: work in progress, see paper for more
Related Work

• **Static program analysis**
  - Simulates the execution of program paths
  - Uses abstraction:
    • often “over-approximate” abstractions
    • Hence imprecision triggers false alarms!

• **Micro execution: locality but with precision**
  - Concrete execution: testing
  - No false alarms due to abstraction (since NO abstraction)
  - Only cause of false alarms: lack of environment assumptions
    • Micro execution may start in an unrealistic initial state
Other Related Work

• Automatic test-driver generation ("closing" open systems)
  - Through static program transformations (PLDI’98, etc.)
  - Automatic static input-interface discovery and test gen (DART,...)

• Automatic dynamic test generation
  - SAGE, Pex, KLEE, S2E, etc.
  - API specific or need test driver with “symbolic” inputs ("param. unit tests")

• Automatic sub-component mock/stub/shim creation
  - Still requires a run-time environment
  - Orthogonal and complementary to micro execution

• How to specify input preconditions and output postconditions
  - Test driver, Code Contracts,...
  - Memory policy = “abstract” test driver - how to edit & refine mem. policies?

• Etc. (see paper)
Conclusion

Micro Execution is the ability to run any code fragment without a user-provided test driver or input data

- Key: a runtime environment which can intercept and redirect input/output memory operations before they occur, and can provide input values according to general rules
- MicroX = 1st VM for test isolation and generation
- Can start/stop executions anywhere and enables local, fast, precise, dynamic analysis of small code fragments & executions
- Lowers the cost of test setup (no test driver)
- How to get the best of static and dynamic program analysis
  - Speed/locality of static analysis with precision of dynamic analysis
  - Enables automatic program decomposition, compositional testing,…
- Many potential applications - but what is the “killer app”?