Building Advanced Coverage-guided Fuzzer for Program Binaries

NGUYEN Anh Quynh <aquynh@gmail.com>
WEI Lei

17/11/2017

Zero Nights, Moscow 2017

Self-introduction

- NGUYEN Anh Quynh, PhD <aquynh @ gmail.com>
  - Nanyang Technological University
  - Operating System, Virtual Machine, Binary Analysis, etc
  - Reverse trilogy: Capstone, Unicorn & Keystone
- WEI Lei, PhD
  - Nanyang Technological University
  - ~ 60 CVEs in Adobe, Apple, PHP etc
  - ~ 50 bug bounties from iDefense VCP, TippingPoint ZDI, and HackerOne.

Agenda

- Coverage-guided fuzzer
  - Background
  - Issues of public guided fuzzers
- Darko fuzzer
  - Features
  - Design & Implementation
- Demo & bugs found
- Conclusions

Coverage-guided Fuzzer

Fuzzer

- Automated software testing technique to find bugs
  - Feed craft input data to the program under test
  - Monitor for errors like crash/hang/memory leaking
  - Focus more on exploitable errors like memory corruption, info leaking
- Maximize code coverage to find bugs
- Blackbox fuzzing
- Whitebox fuzzing
- Graybox fuzzing

Coverage-guided fuzzer

- Instrument target binary to collect coverage info
- Mutate the input to maximize the coverage
- Repeat above steps to find bugs
  - Proved to be very effective
    - Easier to use/setup & found a lot of bugs
  - Trending in fuzzing technology
    - American Fuzzy Lop (AFL) really changed the game

Public guided fuzzers

- AFL
  - Requires source code for instrumentation build
  - Supports *nix binary via emulation mode (Qemu)
- AFL-Cygwin
  - AFL ported to Windows via Cygwin
  - Slow, buggy & development stalled
- WinAFL
  - Windows fork – needs persistent mode support
- AFL-Dyninst
  - Static-based instrumentation struggle on complicated binaries
  - No Windows support

Problems of public guided fuzzers

- Poor support for fuzzing binary
  - AFL emulation mode based on QEMU is limited
    - Only support Linux
    - Limitation of QEMU user mode emulation
  - Only WinAFL handles Windows closed source binaries
- Tricky to use
  - WinAFL persistent mode is really painful
- Suffer on performance & stability
  - DynamoRio is slow & fails to work on some large binaries
  - Needs persistent mode to perform well

DARKO Fuzzer

Darko design

- Motivation: no coverage-guided fuzzer for Windows (Dec 2015)
- Fork AFL fuzzing code & ported to Windows (Apr 2016)
  - Rewrite to work with our target instrumentation
- Support closed source binary for all platforms & architectures
  - To have a cross-platform/architecture fuzzer
  - Build our own instrumentation from scratch (Apr 2016)
    - Replaced with SKORPIO – multi-arch / platform (2017)
  - Support selective binary fuzzing
  - Support persistent mode
- Various other enhancements to AFL (2017)

Darko features

- Pure software-based
- Cross-platform/architecture
  - Native compiled (MSVC on Windows, GCC/Clang on *nix)
- Binary support
  - Full & selective binary fuzzing + Persistent mode
- Fast + stable
  - Stable & support all kind of binaries
  - Order of magnitude faster than DBI/Emulation approaches

Darko implementation - Overview

- AFL-compatible instrumentation
- PoC: AFL-Cygwin + PIN Probe mode (Apr 2016)
  - Applicable to user-space 32-bit Windows binaries
  - Flexible test case post-processor
  - Found bugs in Adobe Reader, Windows Journal, etc
- Static analysis + dynamic binary rewriting (SMC)
  - Speed much better than full binary DBI
- Near native execution speed, ASLR / threading compatible
- Support Windows, Linux & MacOS
- Support for non-X86 architectures underway

Challenges in static analysis

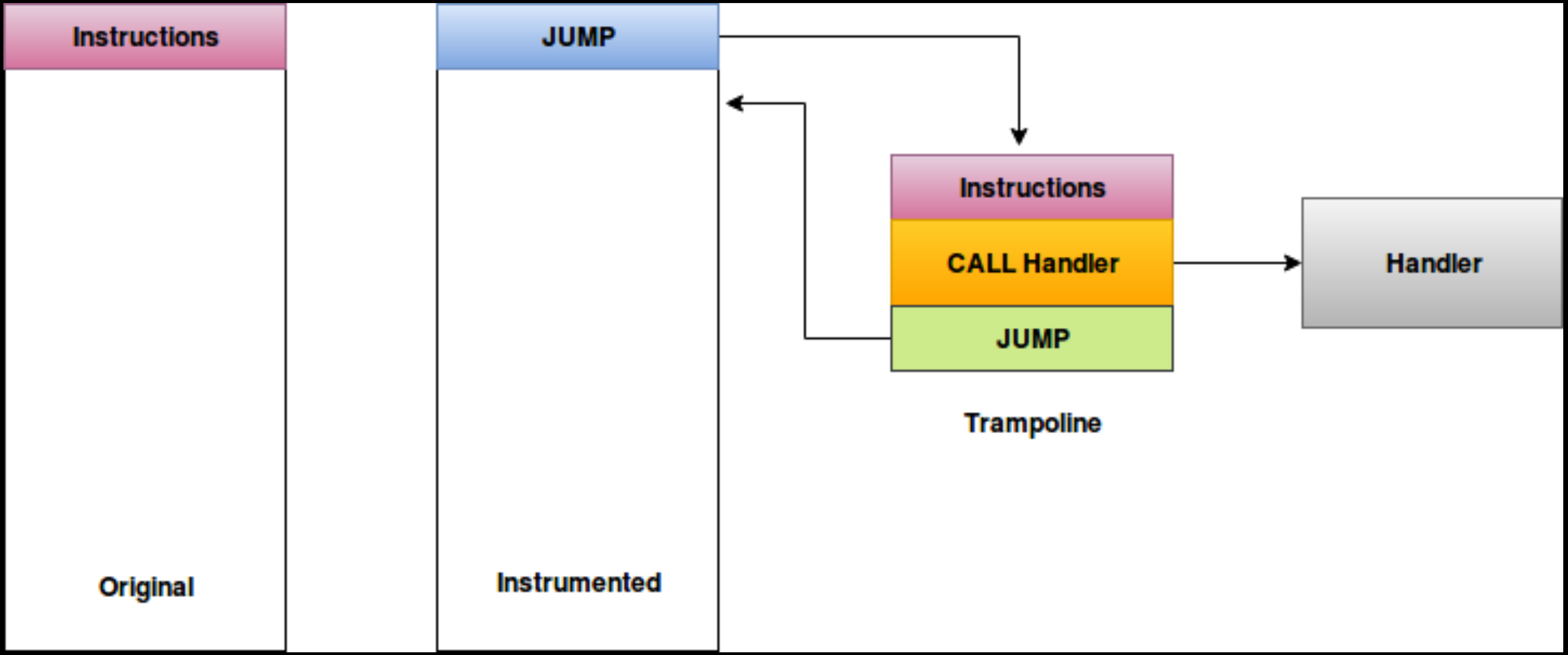
- CFG recovery: correctness v.s. completeness
  - Differentiate data (globals, vftables, jump tables) from code
  - Current effective instrumentation rate: > 60%
  - Rely on IDA Pro to handle compilers & optimizations
- Scalability
  - Tested & works well on Adobe Reader modules (< 10MB - 30MB)
  - For certain compilers, still have FP+ (e.g., mshtml.dll, ~25MB)

Instrumentation

Dynamic instrumentation

- Lesson learned from fuzzers based on DBI (Pin/DynamoRio)
  - Unstable & unreliable
  - Limitation on platforms & architectures
  - Poor performance
    - Cannot do selective instrumentation
- Hooking based mechanism
  - Lightweight & selective
  - Offline analysis on where to instrument
    - Handled with static analysis (beforehand)

Dynamic instrumentation (2)



SKORPIO instrumentation engine

- Cross-platform: Windows, MacOS, Linux, BSD, etc
- Cross architecture: X86, ARM, ARM64, Mips, PowerPC, Sparc
- Multi-level
  - Userspace & OS kernel
  - Instruction level (vs typical function-entry-only)
- Lightweight
  - Implemented in pure C, focus on low-level hooking mechanism
  - Super fast: can be 100x faster than available public hooking frameworks - thanks to many optimization

SKORPIO engine (2)

- Decode instructions at hooking place
  - Use Capstone disassembler (X86, ARM, ARM64, Mips, Sparc, PPC, ...)
- Binary rewrite on code relocation
  - Use Keystone assembler (X86, ARM, ARM64, Mips, Sparc, PPC, …)
- Install user-provided callback at instrumentation hook
- Enable customization/optimization for all requirements
  - Hooking types (JMP or CALL, RET or naked callback)
  - Trampoline setting
  - Thread & internal memory management (OS-agnostic)

Windows instrumentation

- Inject instrumentation into target binary
  - Instrumentation comes in DLL form
  - DLLMain() runs before main program
- Considered Dynamic DLL injection, but rejected
  - Not portable
- Static inject DLL file into target binary
  - Analyze target PE file to locate Sections & Import Directory
  - Append 1 section to relocate Import Directory
  - Point Import Directory Table to the new appended section
    - Append a new entry for injected DLL

Linux & MacOS instrumentation

- LD_PRELOAD to dynamically inject instrumentation
  - Take place before main program runs
  - Linux: shared object file (.so)
  - MacOS: dynamic library (.dylib)
- Inject all instrumentation at initialisation time
  - Can be up to 100k hooks, so must do as quickly as possible
- Inject forkserver at program entry-point, so it takes over later

Detect heap memory corruption

- Windows
  - Enable PageHeap for fuzzing target
  - Low-level exception handling from Windows core
- MacOS & Linux
  - Built-in memory debugging for better control & performance
    - Overload malloc(), free() & co
    - Utilize MMU to detect overflow/underflow errors
      - Off-by-one error
    - Use-after-free error

Demo & bugs found

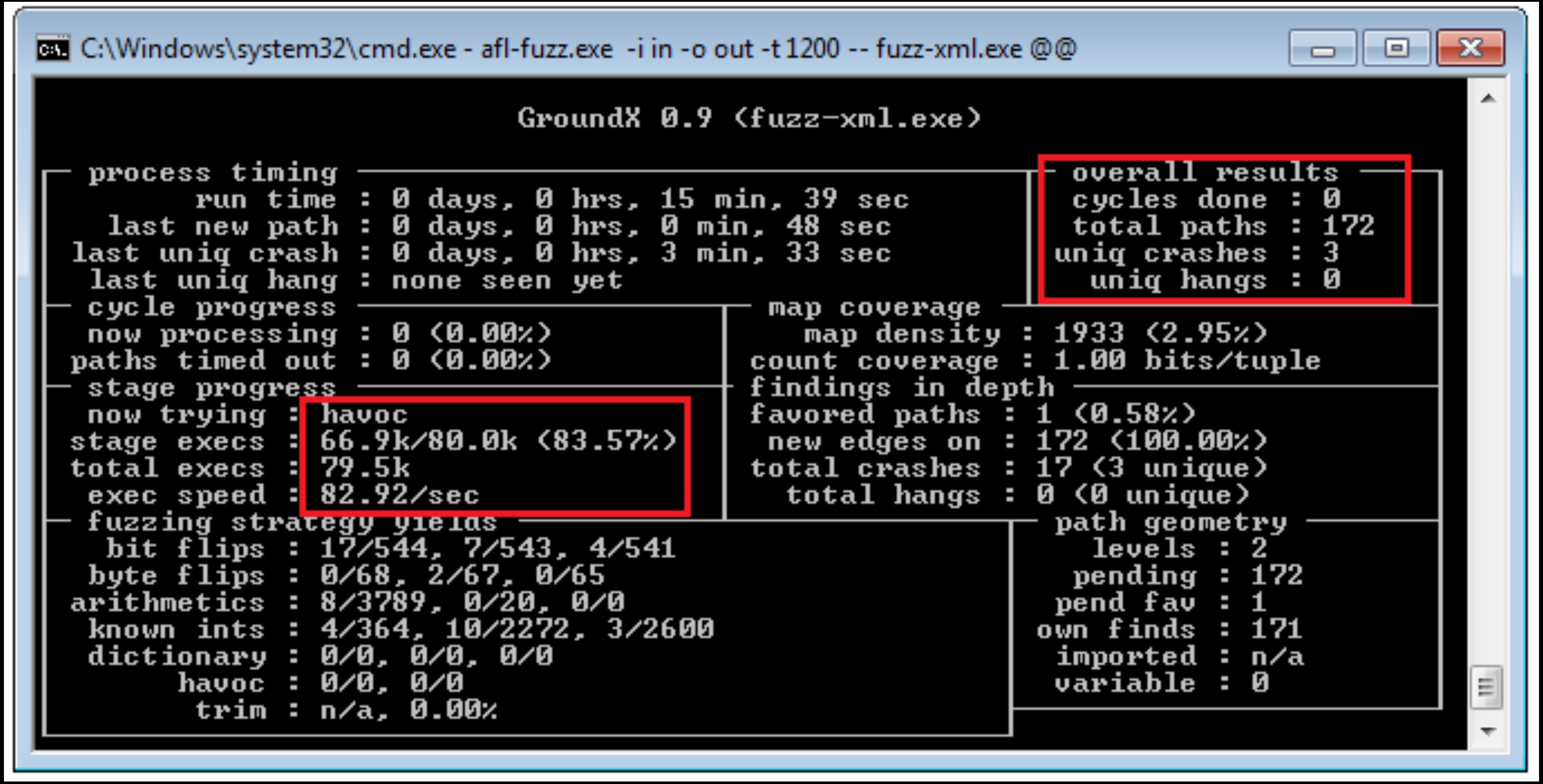
Some results

- PoC (Apr 2016): AFL-Cygwin + Intel PIN probe
  - Adobe Reader U3D: 2 unique bugs in 10 hours
    - CVE-2016-1116: Adobe Reader DC U3D e3_node OOB Access Vulnerability
    - Able to quickly rediscover 12+ bugs on older version:
      - CVE-2014-0523: Adobe Reader U3D Model Node Arbitrary Free Vulnerability
      - CVE-2014-0565: Adobe Reader U3D Line Set Continuation Memory Corruption
      - CVE-2014-9165: Adobe Reader U3D New Object Block Use-after-Free Vulnerability
      - CVE-2015-5586: Adobe Reader U3D Node Blocks Arbitrary Free Memory Corruption
      - CVE-2015-6683: Adobe Reader U3D Bone Weight Modifier Use-after-Free Vulnerability
      - CVE-2016-0933: Adobe Reader DC U3D Bone Weight Modifier OOB Access Vulnerability
      - CVE-2016-1037: Adobe Reader DC U3D Line Set Continuation OOB Access Vulnerability
      - More … (will release the repros on GitHub)
  - Libxml2-2.7.8.win32 – 10 unique bugs in a week
  - Windows Journal – some bugs

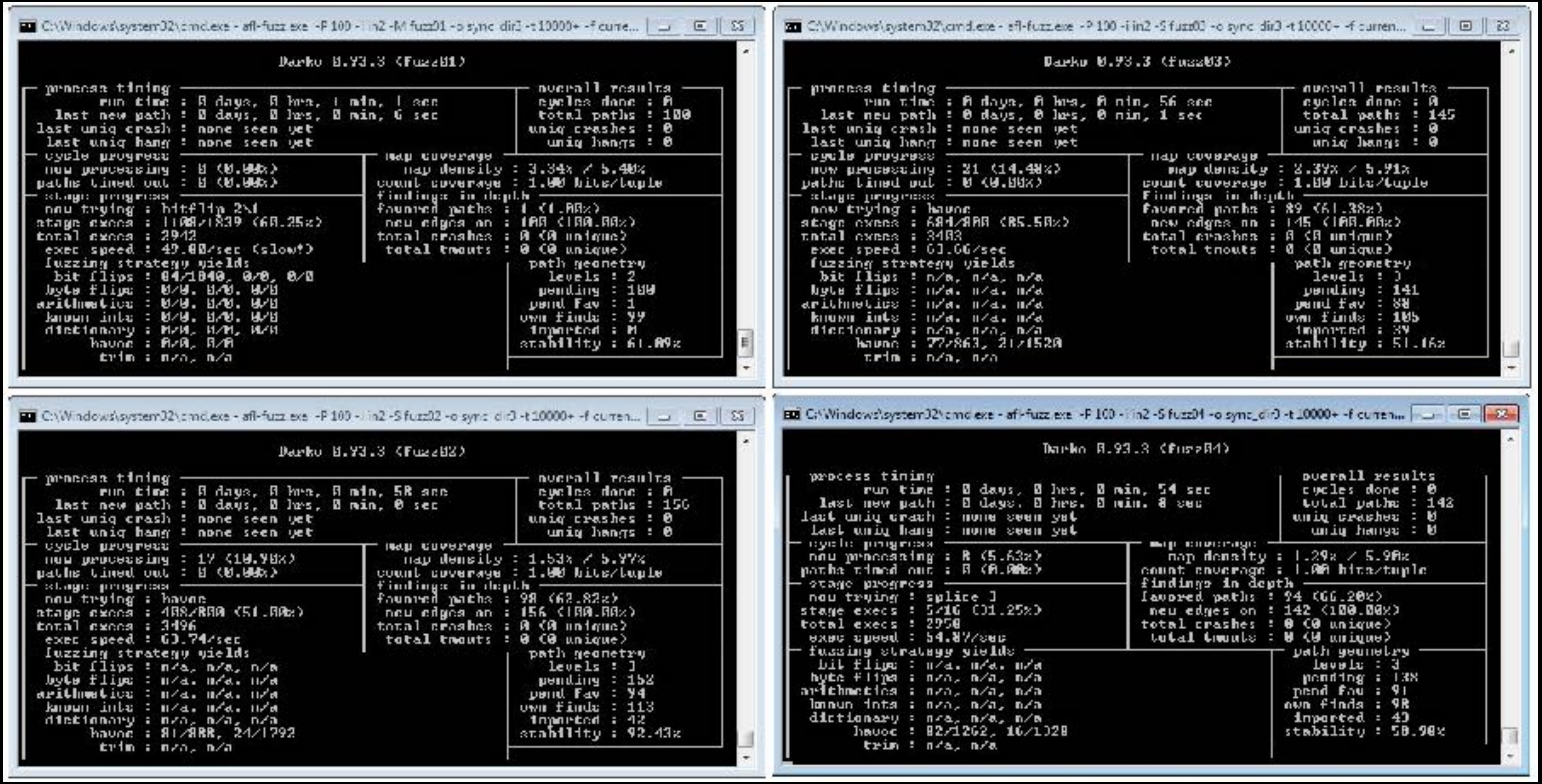
Experiments

- Libxml2 – native, not compatible with persistent mode
- Native run with persistent mode:
  - UnRAR – persistent mode + parallel fuzzing
  - Msxml6 – persistent mode + parallel fuzzing
  - Adobe Reader – Javascript engine
  - Adobe Reader 3D

Demos - libxml2.dll



Demos - unrar.exe



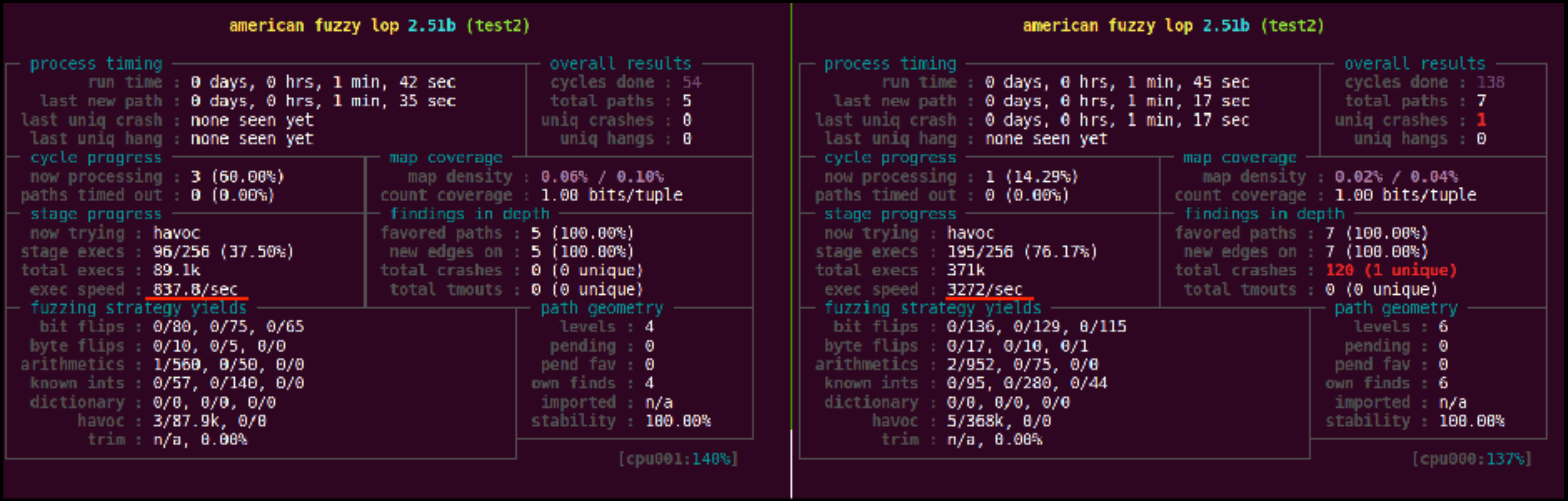
Demos - afl-tmin.exe

```
λ afl-tmin.exe -t 1000 -i id_000002_11 -o id_000002_11.trim -- fuzz-xml.exe @@ 2>nul
afl-tmin.exe 2.49b by Nguyen Anh Quynh, 2017
Based on AFL 2.49b by <lcamtuf@google.com>

[+] Read 68 bytes from 'id_000002_11'.
[*] Performing dry run (timeout = 1000 ms)...
[+] Program terminates normally, minimizing in instrumented mode.
[*] Stage #0: One-time block normalization...
[+] Block normalization complete, 60 bytes replaced.
[*] --- Pass #1 ---
[*] Stage #1: Removing blocks of data...
    Block length = 4, remaining size = 68
    Block length = 2, remaining size = 36
    Block length = 1, remaining size = 35
[+] Block removal complete, 33 bytes deleted.
[*] Stage #2: Minimizing symbols (8 code points)...
[+] Symbol minimization finished, 1 symbol (1 byte) replaced.
[*] Stage #3: Character minimization...
[+] Character minimization done, 0 bytes replaced.
[*] --- Pass #2 ---
[*] Stage #1: Removing blocks of data...
    Block length = 2, remaining size = 35
    Block length = 1, remaining size = 35
[+] Block removal complete, 0 bytes deleted.

     File size reduced by : 48.53% (to 35 bytes)
    Characters simplified : 174.29%
     Number of execs done : 72
          Fruitless execs : path=46 crash=0 hang=0

[*] Writing output to 'id_000002_11.trim'...
[+] We're done here. Have a nice day!
```

Demos - afl-analyze.exe

```
λ afl-analyze.exe -q -t 1000 -i note.xml -- fuzz-xml.exe @@
afl-analyze.exe 2.49b by Nguyen Anh Quynh, 2017
Based on AFL 2.49b by <lcamtuf@google.com>

[+] Read 123 bytes from 'note.xml'.
[*] Performing dry run (timeout = 1000 ms)...
[*] Analyzing input file (this may take a while)...

    01 - no-op block                01 - suspected length field
    01 - superficial content        01 - suspected cksum or magic int
    01 - critical stream            01 - suspected checksummed block
    01 - "magic value" section

[000000] <   ?   x   m   l   #20 v   e   r   s   i   o   n   =   "   1
[000016] .   0   "   >   #0d #0a <   n   o   t   e   >   #0d #0a #20 #20
[000032] <   t   o   >   T   o   v   e   <   /   t   o   >   #0d #0a #20
[000048] #20 <   f   r   o   m   >   J   a   n   i   <   /   f   r   o   >
[000064] m   >   #0d #0a #20 #20 <   b   o   d   y   >   D   o   n   '
[000080] t   #20 f   o   r   g   e   t   #20 m   e   #20 t   h   i   s   >
[000096] #20 w   e   e   k   e   n   d   !   <   /   b   o   d   y   >
[000112] #0d #0a <   /   n   o   t   e   >   #0d #0a

[+] Analysis complete. Interesting bits: 69.92% of the input file.
[+] We're done here. Have a nice day!
```

Demos - AFL -Q (Linux) vs Darko



Demos - Darko vs AFL native on MacOS

```
                  american fuzzy lop 2.51b (test1)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 1 min, 37 sec       │  cycles done : 21     │
│   last new path : 0 days, 0 hrs, 1 min, 16 sec       │  total paths : 7      │
│ last uniq crash : 0 days, 0 hrs, 1 min, 30 sec       │ uniq crashes : 1      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 6 (85.71%)        │    map density : 0.00% / 0.01%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 7 (100.00%)           │
│ stage execs : 40/512 (7.81%)        │  new edges on : 7 (100.00%)           │
│ total execs : 102k                  │ total crashes : 42 (1 unique)         │
│  exec speed : 1010/sec              │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/144, 0/137, 1/123                    │    levels : 5          │
│  byte flips : 0/18, 0/11, 0/2                        │   pending : 0          │
│ arithmetics : 2/1008, 0/74, 0/0                      │  pend fav : 0          │
│  known ints : 0/101, 0/308, 0/88                     │ own finds : 6          │
│  dictionary : 0/0, 0/0, 0/0                          │  imported : n/a        │
│       havoc : 4/90.0k, 0/10.5k                       │ stability : 100.00%    │
│        trim : 55.56%/2, 0.00%                        ├────────────────────────┘
└──────────────────────────────────────────────────────┘          [cpu: 93%]
```

```
                  american fuzzy lop 2.51b (test2)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 1 min, 35 sec       │  cycles done : 81     │
│   last new path : 0 days, 0 hrs, 0 min, 32 sec       │  total paths : 7      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 17 sec       │ uniq crashes : 1      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 6 (85.71%)        │    map density : 0.02% / 0.04%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 7 (100.00%)           │
│ stage execs : 736/2048 (35.94%)     │  new edges on : 7 (100.00%)           │
│ total execs : 115k                  │ total crashes : 14 (1 unique)         │
│  exec speed : 1158/sec              │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/136, 0/129, 0/115                    │    levels : 5          │
│  byte flips : 0/17, 0/10, 0/1                        │   pending : 0          │
│ arithmetics : 2/952, 0/50, 0/0                       │  pend fav : 0          │
│  known ints : 0/96, 0/280, 0/44                      │ own finds : 6          │
│  dictionary : 0/0, 0/0, 0/0                          │  imported : n/a        │
│       havoc : 5/112k, 0/0                            │ stability : 100.00%    │
│        trim : 20.00%/1, 0.00%                        ├────────────────────────┘
└──────────────────────────────────────────────────────┘          [cpu: 93%]
```

Demos - MacOS

Conclusions

- DARKO is an advanced coverage-guided fuzzer
  - Pure software-based
  - Cross-platform/architecture
  - Binary support
    - Fuzz full binary + Persistent mode
  - Fast + stable

- SKORPIO engine will be released to public in near future

Questions?

Building Advanced Coverage-guided Fuzzer for Program Binaries

NGUYEN Anh Quynh <aquynh@gmail.com>

WEI Lei