Code: analysis, bugs, and security

supported by Bitdefender

Fuzzing and symbolic execution

Marius Minea

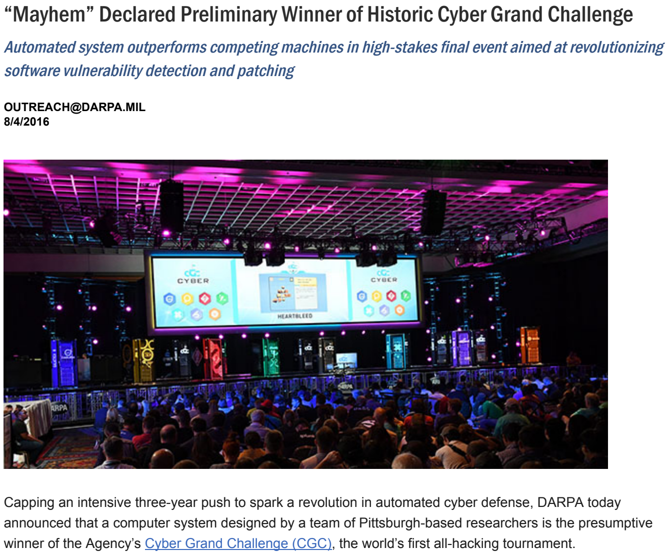
marius@cs.upt.ro

22 November 2017

A long time ago ...

```
                  .oO Phrack 49 Oo.

             Volume Seven, Issue Forty-Nine

                      File 14 of 16

          BugTraq, r00t, and Underground.Org
                       bring you

        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        Smashing The Stack For Fun And Profit
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

                    by Aleph One
                aleph1@underground.org
```

Now: Cyber Grand Challenge



“Mayhem” Declared Preliminary Winner of Historic Cyber Grand Challenge

Automated system outperforms competing machines in high-stakes final event aimed at revolutionizing software vulnerability detection and patching

OUTREACH@DARPA.MIL
8/4/2016

Capping an intensive three-year push to spark a revolution in automated cyber defense, DARPA today announced that a computer system designed by a team of Pittsburgh-based researchers is the presumptive winner of the Agency’s Cyber Grand Challenge (CGC), the world’s first all-hacking tournament.

What does it take?

Then: intuition, creativity, a debugger

Now: debugger not enough

lots of *math*: constraint / satisfiability checking

precise modeling of instruction semantics (specialized platforms)

intelligent *combination* of different techniques

engineering skills for performance

From bug finding to automatic exploit generation

Fuzzing
lightweight technique, evolves inputs
aims for input variety, high statement/branch coverage

Symbolic execution
more expensive, analyzes program control flow
attempts path coverage

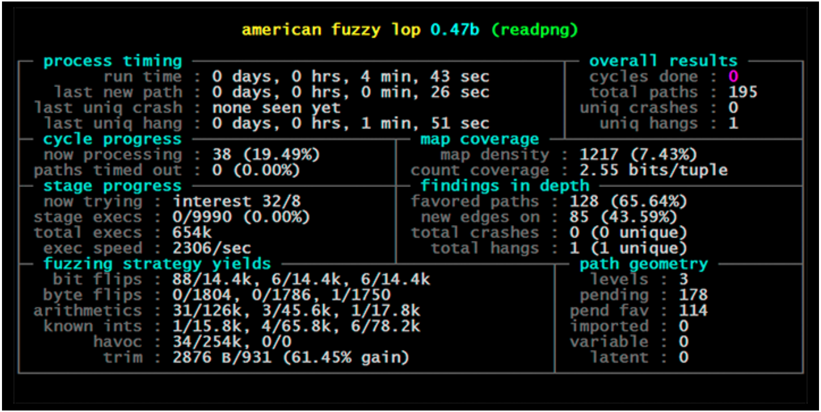
Automatic exploit generation
find path to bug, then synthesize exploit

Fuzzing

Fuzzer case study: AFL (American Fuzzy Lop)

fuzzer by Michal Zalewski, `http://lcamtuf.coredump.cx/afl/`

active development, scores of bugs found in key software



```
                    american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec       │  total paths : 195    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec       │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 38 (19.49%)       │    map density : 1217 (7.43%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.55 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 32/8         │ favored paths : 128 (65.64%)           │
│ stage execs : 0/9990 (0.00%)        │  new edges on : 85 (43.59%)            │
│ total execs : 654k                  │ total crashes : 0 (0 unique)           │
│  exec speed : 2306/sec              │   total hangs : 1 (1 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k            │    levels : 3          │
│  byte flips : 0/1804, 0/1786, 1/1750                │   pending : 178        │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k             │  pend fav : 114        │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k             │  imported : 0          │
│       havoc : 34/254k, 0/0                          │  variable : 0          │
│        trim : 2876 B/931 (61.45% gain)              │    latent : 0          │
└─────────────────────────────────────────────────────┴────────────────────────┘
```

AFL: Basic Workings

if source available: compiles project with coverage instrumentation
  (gcc/g++/clang wrapper)

binary-only: execute under QEMU in user-mode emulation

start: small set of initial test inputs to evolve

workings:
  maintains queue of test inputs
  mutates inputs using several strategies
  if *new coverage* achieved, add mutant to input queue

  minimize each test input (keeping coverage)
  minimize input corpus (avoids overlap)

AFL: Measuring coverage

Goal: distinguish “interesting” basic-block traces

Example: `A -> B -> C -> D -> E`
and `A -> B -> D -> C -> E`
have different transition pairs `(C, D)` and `(D, C)`

transition coverage provides more info than basic block coverage
also self-loops `A->A` (tight program loops)

can’t record exhaustively ⇒ do some hashing for compression

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

AFL keeps 64kB map of branch pairs
⇒ $<$ 14% collision on 20k branches

Detecting new behaviors

1) *new tuples* (of basic blocks) in branch map

2) coarse *hit count* of branch tuples

don't keep actual counts, just ranges (buckets)
1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+

fast to compute (bit ops: powers of 2)
tracks "interesting" changes
(individual low counts + changes between intervals)

Evolving input queue

For most targets, keep 1k – 10k entries (test inputs)
10-30% due to new tuple discovery
rest: changes in hit counts

Culling the test corpus
periodically find subset which cover all branch tuples
prioritize based on execution latency, file size
when generating mutants, use “favored” entries 90%+ of the time

Trimming input files (for size)
  affects performance (execution time)
  affects mutation (effect of change more likely in small input)
⇒ try to remove blocks of data from input, check if coverage kept

Fuzzing strategies

Deterministic

Sequential bit flips: flip 1-4 bits, stepping one bit at a time
yield: 70 (single flip) downto 10 new paths per million
expensive (one execve() for each bit of input)
Sequential byte flips (1-4 bytes)
Simple arithmetic: incr/decr integer values (small inc, ±35)
Known integers: can trigger edge conditions in typical code
(-1, 256, 1024, `MAX_INT`, etc)

Nondeterministic: stacked bit flips, insertions, deletions

Test case splicing

take two inputs differing in ≥ 2 places, splice at some midpoint,
then do nondeterministic tweaks
usually $+20\%$ of execution paths

Grammars and keywords

To fuzz structured input, can start with dictionary of keywords

Even random keyword combinations yield interesting valid SQL

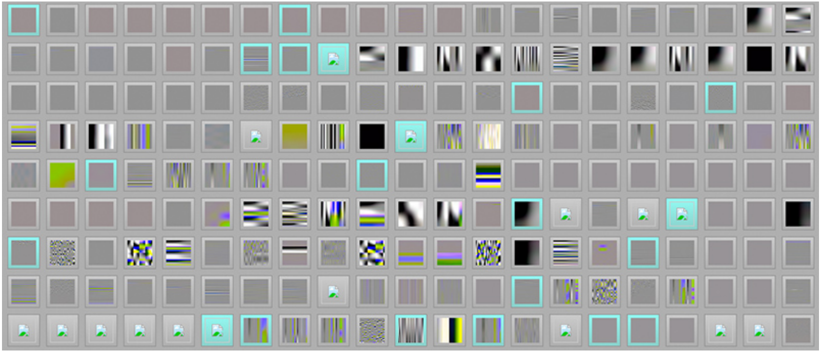
```
select sum(1)LIMIT(select sum(1)LIMIT -1,1);
select  round( -1)‘‘‘‘;
select length(?)in( hex(1)+++1,1);
select abs(+0+ hex(1)-NOT+1) t1;
select DISTINCT "Y","b",(1)"Y","b",(1);
```

Can also automatically find keywords by detecting
which walking byte flips trigger a new execution path

AFL can synthesize complex file structures (e.g. images)
even when starting from invalid input!

Synthesizing JPEGs from scratch

start with text file containing 'hello'
fuzzer finds coverage change with markers 0xFF, 0xF9, etc.

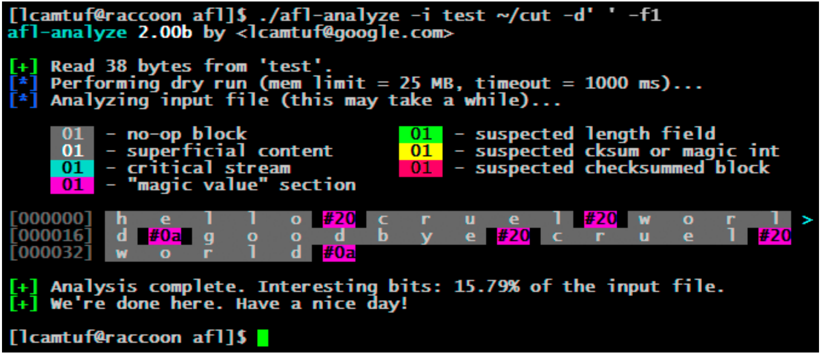


`https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html`

six hours to generate first image, then others in rapid sequence
⇒ general-purpose fuzzing works; improved if format-specific

Automatic format detection

`afl-analyze` tool attempts to classify bytes of input:



```
[lcamtuf@raccoon afl]$ ./afl-analyze -i test ~/cut -d' ' -f1
afl-analyze 2.00b by <lcamtuf@google.com>

[+] Read 38 bytes from 'test'.
[*] Performing dry run (mem limit = 25 MB, 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] h   e   l   l   o   #20 c   r   u   e   l   #20 w   o   r   l   >
[000016] d   #0a g   o   o   d   b   y   e   #20 c   r   u   e   l   #20
[000032] w   o   r   l   d   #0a

[+] Analysis complete. Interesting bits: 15.79% of the input file.
[+] We're done here. Have a nice day!

[lcamtuf@raccoon afl]$
```

Classification of input fields

"No-op blocks": no apparent control flow change
(data payload)
"Superficial content": some control flow changes
(strings in rich documents)
"Critical stream": control flow altered in correlated ways
(keywords, magic values, compressed data)
"Suspected length field" – small int causing control flow change
"Suspected cksum or magic int"
"Suspected checksummed block"
"Magic value section"

Performance: fork server

For libraries, usual fuzzing approach is with a simple client program
but: overhead for execve(), linker, library initialization routines

Idea: modify binary to stop after all initialization, before main code

On command from fuzzer, fork() clone of already-loaded program
  fast due to copy-on-write

Symbolic execution

Symbolic execution

described since mid-seventies (James C. King 1976, others)

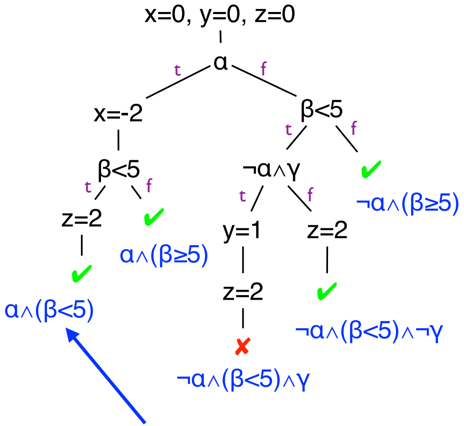
program is executed by a special interpreter, using *symbolic* inputs
⇒ results in symbolic execution tree
  each branch: *path condition* as formula over symbolic variables
  tree traversal stops when path condition becomes unsatisfiable

Can be used to:
  attaining high coverage
  or try to reach a specific branch

Successful mature technique, hundreds of papers, many tools:
Java Pathfinder, (j)CUTE, CREST, KLEE, Pex, SAGE, ...
  for C/C++, C#, Java, more recently JavaScript

Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



path condition

slide: Jeff Foster, UMD

Constraint solving in symbolic execution

Symbolic execution: improved by advances in satisfiability checking (fundamental problem in logic)

here: *satisfiability modulo theories*
  incorporates knowledge specific to type of formula:
    linear integer/real arithmetic, bitvectors, arrays, strings

Annual SMT competition, continuous advances in performance (millions of constraints for pure boolean formulas)

Solvers: Z3 (Microsoft Research), CVC (NYU), STP (Stanford), Yices (SRI), etc.
  most open-source

Concolic (concrete + symbolic) execution

symbolic execution is directed by *concrete* run

keep variable symbolic if possible, else fall back to concrete values
  native functions, nonlinear arithmetic, library/system functions

```
y = hash(x);  // can't solve hash => y becomes concrete
if (x + y > 0)
  // path 1
else
  // path 2
```

Assume: `x = 20; y = hash(20) = 13` ⇒ reach *path 1*

To reach *path 2*, negate $x + y > 0$, with *concrete* y (y = 13)

Solver might return, e.g., x = -15
  if lucky, -15 + hash(-15) < 0, we reach path 2
  else execution still follows path 1, retry

⇒ worst-case: degrades to *random testing*

KLEE: symbolic execution for LLVM

Cadar, Dunbar, Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008 (best paper)

90%+ coverage on coreutils + busybox

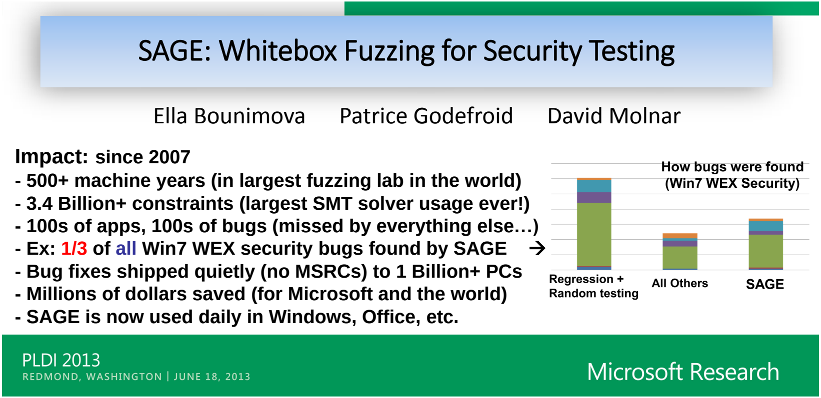
56 serious bugs in 430 kloc, some bugs 15 years old
simple crash inputs generated for several programs

based on LLVM infrastructure (analyzes LLVM bitcode)

lots of engineering work

path exploration heuristics
efficient branching due to copy-on-write
models for library functions, file system, etc.

Symbolic execution in industry



SAGE: Whitebox Fuzzing for Security Testing

Ella Bounimova   Patrice Godefroid   David Molnar

Impact: **since 2007**

- **500+ machine years (in largest fuzzing lab in the world)**
- **3.4 Billion+ constraints (largest SMT solver usage ever!)**
- **100s of apps, 100s of bugs (missed by everything else…)**
- **Ex: 1/3 of all Win7 WEX security bugs found by SAGE →**
- **Bug fixes shipped quietly (no MSRCs) to 1 Billion+ PCs**
- **Millions of dollars saved (for Microsoft and the world)**
- **SAGE is now used daily in Windows, Office, etc.**

PLDI 2013
REDMOND, WASHINGTON | JUNE 18, 2013

Microsoft Research

From vulnerabilities to exploits

Which bugs are exploitable?

Easy to find functions which are *surely* unsafe

For cases which are *potentially* be unsafe, must decide
1) is it really a bug ?
2) can it be exploited ?

Automated Exploit Generation

S. Heelan, Automatic generation of control flow hijacking exploits for software vulnerabilities, MSc thesis, Oxford, 2009

Two steps:
generate input that executes and exploitable program path
express conditions necessary to transfer control to shellcode

Avgerinos, Brumley et al.:
  Automatic Exploit Generation, NDSS 2011
  Unleashing Mayhem on Binary Code, IEEE S&P 2012

applied large-scale to Debian code
can generate buffer overflow and format string attacks
(form constraints on symbolic instruction pointer / format string)

checking Debian for exploitable bugs

37,000 programs

16 billion verification queries

~$0.28/bug
~$21/exploit

test cases

2,606,000 crashes

14,000 unique bugs

152 *new* exploits

* [ARCB, ICSE 2014, ACM Distinguished Paper], [ACRSWB, CACM 2014]

slide: David Brumley, CMU, 2015

Combining techniques

Driller: Augmenting Fuzzing Through Selective Symbolic Execution
Stephens, Kruegel, Vigna et al. (UC Santa Barbara), NDSS 2016

Key insight: fuzzing is cheap, good overall coverage

Symbolic execution: expensive, path explosion,
but can pass through precise, complex condition

Sample code

```
int check(char *x, int depth) {
  if (depth >= 100) {
    return 0;
  } else {
    int count = (*x == ’B’) ? 1 : 0;
    count += check(x+1, depth+1);
    return count;
  }
}

int main(void) {
  char x[100];
  read(0, x, 100);

  if (check(x, 0) == 25)
    vulnerable();
}
```

Listing 4. A program that causes a path explosion under concolic execution.

Driller: Structure of explored call graph

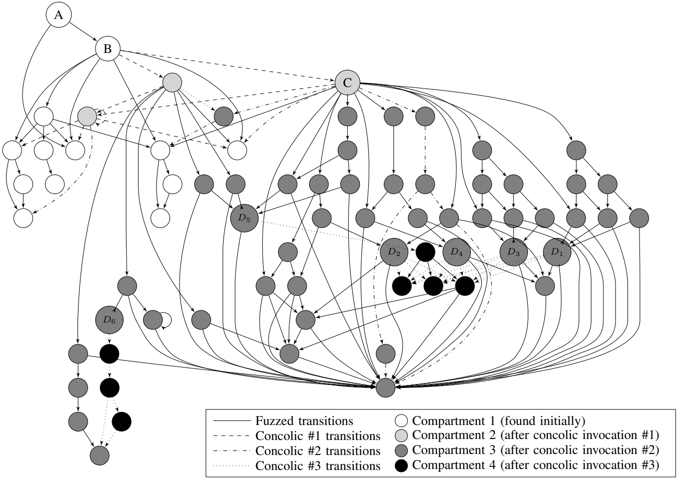


Fig. 10. Graph visualizing the progress made by Driller in discovering new compartments. Each node is a function; each edge is a function call, but return edges are excluded to maintain legibility. Node "A" is the entry point. Node "B" contains a magic number check that requires the symbolic execution component to resolve. Node "C" contains another magic number check.

Path to vulnerability

Fuzzing helps explore a “compartment” efficiently

Symbolic execution finds “door” between compartments

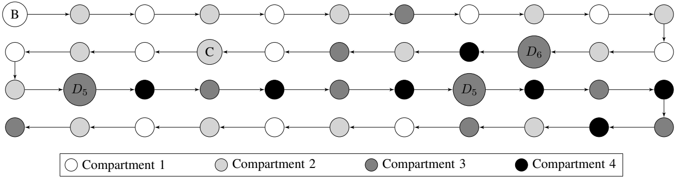


Fig. 11. The sequence of compartments through which execution flows for a trace of the crashing input for CGC application 2b03cf01. Driller’s ability to “break into” the fourth compartment (represented by the black nodes) was critical for generating the crashing input. The generated, derandomized crashing input was “A\x00\x00\x00\x00\x00\x00\x00\x9c6\x00\x00\x18\x04\x00\x00\x18’\x00\x00A\x00\x00\x00\x00\x00\x00\x00\x9c6\x00\x00\x19\x04\x00\x00\x14\x00\x00\x00A\x00\xf8\xff\xff\xec\x00d\x96X\x0c\x00\x06\x08\x00\x00\x10\x00\x00\x00A\x00\x00\x00\x00\x00\x00\xfb\x96X\x0c\x00\x02\x08\x00\x00\x18’\x00\x00A\x00\xebA\x00\x00d\x96X\x0c\x00\x06”. The full exploit specification, conforming to the DARPA CGC exploit specification format and accounting for randomness, is available in Appendix A.

Fuzzing vs. concolic execution

G. Case Study

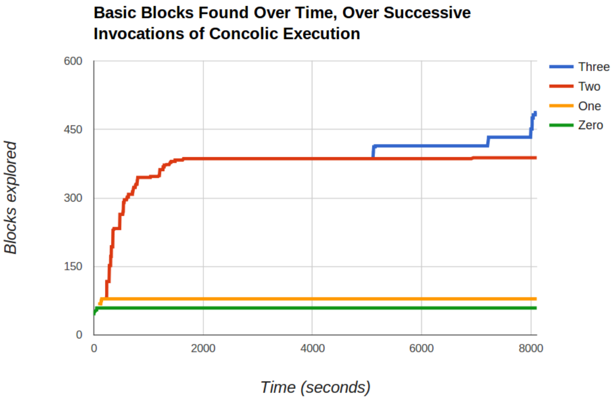
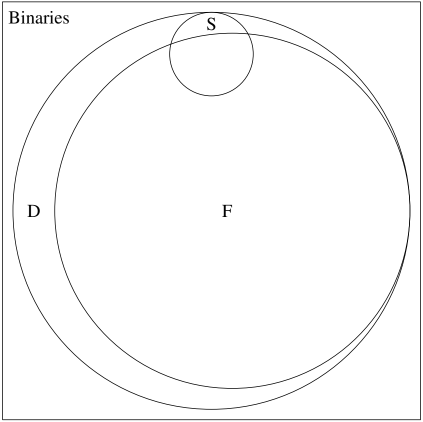


Fig. 9. For the binary 2b03cf01, which Driller crashed in about 2.25 hours, this graph shows the number of basic blocks found over time. Each line represents a different number of invocations of symbolic execution from zero to three invocations. After each invocation of symbolic execution, the fuzzer is able to find more basic blocks.

Bugs by technique



Method	Crashes Found
Fuzzing	68
Fuzzing ∩ Driller	68
Fuzzing ∩ Symbolic	13
Symbolic	16
Symbolic ∩ Driller	16
Driller	77

Where to from here?

Fascinating work, rapid reaction, spectacular advance
  strong reliance on theory/logic (advances in SMT solvers)
  open-source platforms (angr, BAP, BINSEC, etc.)
  engineering, performance, integration of techniques

More good reads:

Yan Shoshitaishvili, Ruoyu Wang, Ch. Kruegel, G. Vigna et al.:
(State of) The Art of War: Offensive Techniques in Binary Analysis,
IEEE S&P 2016

describes `http://angr.io/` platform from UCSB