Symbolic Security Predicates

Hunt Program Weaknesses

Alexey Vishnyakov @VishnyaSweet
Vlada Logunova
Eli Kobrin
Daniil Kuts
Darya Parygina
Andrey Fedotov @xfedotoffx

December 3, 2021

ISP RAS

arxiv.org/abs/2111.05770

Motivation

- Novel code inevitably brings new bugs and weaknesses
- The security development lifecycle (SDL) improves application quality and defends it from malicious attacks
- Fuzzing is continuously applied to detect crashes during development process
- Advanced hybrid fuzzing benefits from dynamic symbolic execution (DSE) that
  - explores complex program states and
  - automatically detects weaknesses
- We focus on automatic detection for undefined behavior and memory access violation errors
- DSE generates seeds that trigger integer overflow, out-of-bounds access, etc.

Hybrid Fuzzing Setup

- Build target with sanitizers for fuzzer
- Build target without sanitizers for Sydr
- Sydr explores new program states via branch inversion
- Fuzzer takes seeds from Sydr that increase code coverage
- Sydr runs on corpus and generates new seeds that trigger errors
- Generated seeds are verified on sanitizers

Dynamic Symbolic Execution

Dynamic symbolic execution with Sydr:

Sydr = DynamoRIO (The DR. is in.) + TRITON (Dynamic Binary Analysis) + Z3

- Sydr uses DynamoRIO as a DBI framework
- Sydr uses Triton as a DSE engine
- Triton uses Z3 as an SMT solver
- Each input byte is modeled by a free *symbolic variable*
- Instructions interpretation produce SMT formulas
- *Symbolic state* maps registers and memory to SMT formulas
- *Path predicate* contains taken branch constraints
- *Path predicate slicing* removes irrelevant constraints from path predicate

Contributions

- Symbolic function semantics for common C/C++ standard library functions
- Security predicates for undefined behavior and memory access violation errors
- Juliet Dynamic measures dynamic bug detection tools accuracy on Juliet test suite

Function Semantics

- We just skip some functions to increase performance and reduce overconstrainting (`malloc`, `strcpy`, `printf`, etc.)
- Both uppercase and lowercase characters are permissible for `tolower(int ch)`
- However, relying on concrete execution trace ends up in overconstrainting to single letter case
- We always update concrete state via DBI, but we skip symbolic execution of functions
- We propose functions semantic models which can incorporate more symbolic states and speed up the execution:
  $ite(ch - \texttt{'A'} < 26,\ ch - (\texttt{'A'} - \texttt{'a'}),\ ch)$
- Function semantics extend symbolic states and assist bug detection
- Moreover, we can perform function level security checks

String Comparison

- Character search: `memchr`, `strchr`, `strstr`, `strlen`, etc.
- Lexicographical comparison: `memcmp`, `strcmp`, etc.
- `memcmp(lhs, rhs, count)`:

$$lhs[0] - rhs[0] + \sum_{i=1}^{count-1} (lhs[i] - rhs[i]) * ite\left(\bigwedge_{k=0}^{i-1} lhs[k] = rhs[k], 1, 0\right)$$

String to Integer Conversion

- `strtol`, `strtoul`, `strtoll`, `std::cin`, etc.
- `atoi` and `scanf("%d", &x)` call `strto*l` inside
- We compute in twice bigger bit vector and add constraints $LONG\_MIN \leq x \leq LONG\_MAX$ to overcome overflow

$$\pm (c_n c_{n-1} \ldots c_1 c_0)_b \longrightarrow x \tag{1}$$

$$\begin{aligned} a_k = ite(c_k \geq \texttt{'0'} \wedge \ c_k \leq \texttt{'9'} \wedge \ c_k < \texttt{'0'} + b, \\ c_k - \texttt{'0'}, \\ ite(c_k \geq \texttt{'a'} \wedge \ c_k < \texttt{'a'} + b - 10, \\ c_k - \texttt{'a'} + 10, \ c_k - \texttt{'A'} + 10)) \end{aligned} \tag{2}$$

$$|x| = \sum_{k=0}^{n} a_k b^k, \ x = ite(sign = \texttt{'-'}, -|x|, |x|) \tag{3}$$

$$\begin{aligned} (c_k \geq \texttt{'0'} \wedge \ c_k \leq \texttt{'9'} \wedge \ c_k < \texttt{'0'} + b) \vee \\ (c_k \geq \texttt{'a'} \wedge c_k < \texttt{'a'} + b - 10) \vee \\ (c_k \geq \texttt{'A'} \wedge c_k < \texttt{'A'} + b - 10) \end{aligned} \tag{4}$$

Function Semantics Benchmarking – Path Predicate

Application	Default		Function Semantics	
	Branches	Time	Branches	Time
bzip2recover	5131	6s	5131	6s
cjpeg	8008	19s	6992	18s
faad	470585	21m	466697	15m52s
foo2lava	910737	21m9s	905592	18m20s
hdp	66070	43s	29265	20s
jasper	837643	14m47s	771806	10m37s
libxml2	53400	40s	8873	12s
minigzip	8977	1m4s	8977	1m3s
muraster	7102	5s	4453	4s
pk2bm	3665	2s	658	1s
pnmhistmap_pgm	967187	9m21s	967155	9m2s
pnmhistmap_ppm	7864	12s	7822	11s
readelf	62713	41s	13649	10s
yices-smt2	19352	17s	10340	11s
yodl	8329	9s	5340	5s

Function Semantics Benchmarking – 2-Hour Benchmark

Application	Default				Function Semantics			
	Accuracy	SAT	Queries	Time	Accuracy	SAT	Queries	Time
bzip2recover	100%	2101	5131	47m35s	100%	2101	5131	45m38s
cjpeg	100%	50	2656	120m	100%	50	3750	120m
faad	97.11%	1974	3072	120m	98.91%	1560	2414	120m
foo2lava	87.1%	31	5998	120m	99.02%	205	6668	120m
hdp	76.69%	1171	4122	120m	72.22%	5893	12172	120m
jasper	99.62%	8457	22538	120m	96.61%	9528	24472	120m
libxml2	51.27%	1063	18485	120m	82.44%	1247	8970	5m53s
minigzip	51.47%	7569	8977	16m16s	51.47%	7569	8977	16m16s
muraster	99.94%	3304	6041	120m	100%	360	470	120m
pk2bm	99.45%	183	3664	15m55s	100%	189	657	4m55s
pnmhistmap_pgm	99.99%	19351	28932	120m	100%	19964	29369	120m
pnmhistmap_ppm	99.07%	107	7990	27m26s	99.12%	114	7948	25m31s
readelf	87.38%	1022	9541	120m	85.82%	2363	6541	120m
yices-smt2	73.79%	4258	16222	120m	70.27%	5534	11753	11m5s
yodl	36.25%	1153	9403	51m3s	98.26%	1150	6414	1m50s

Security Predicates

- *Security predicate* for some error type (weakness) is a Boolean predicate that holds true iff the instruction (or function) triggers an error
- We symbolically execute a program with input that doesn't lead to crash
- We construct security predicates that check for undefined behavior and memory access violation
- We conjunct a security predicate with sliced branch constraints from the path predicate, i.e. constraints over symbolic variables that are relevant to variables in security predicate
- If SAT, Sydr reports an error and generates new seed reproducing the error

Supported Security Predicates

- Division by zero
- Null pointer dereference
- Out-of-bounds access
- Integer overflow

Out-of-bounds Access

- We build security predicate at each symbolic pointer dereference (that depends on user input)
- We maintain shadow heap and stack to determine address bounds
- However, both bounds cannot be always determined in binary code
- Sydr can heuristically retrieve the array base from concrete part of symbolic address expression:
  - `[rdx + rax]` – `rax` is concrete array base and `rdx` is symbolic index
- Moreover, Sydr wraps memory copy functions (`memcpy`, `memmove`, `memset`, `strncpy`, etc.) to detect buffer overflows

Integer Overflow

- Integer overflow occurs quite often in binary code
- Checking all these situations slows down analysis and leads to false positives
- *Source* is an instruction where integer overflow may happen
- *Sink* is a place in code where preceding flaw may lead to critical error
- We call solver in error sinks that use potentially overflowed value
  - Conditional branches
  - Memory access addresses
  - Function arguments
- We create security predicates for unsigned (CF) and signed (OF) overflows that are true when the corresponding flag is equal to 1

Signedness Detection

- We detect operation signedness in binary code:
  - Iterate backwards over branch constraints that use variables from sink
  - Conditional branches help to detect signedness (for instance, `jl` is signed branch)
- We can also guess signedness when input data came from `strto*l`

DEMO: Integer Overflow to Buffer Overflow (Juliet Test)

- 32-bit program
- Input: +00000000002
- `strtol` in line 6
- Integer overflow in line 9
- Buffer overflow in line 12
- Solution: +01073741825

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int size;
6      fscanf(stdin, "%d", &size);
7      if (size <= 0) return 1;
8      size_t i;
9      int *p = malloc(size * sizeof(int));
10     if (p == NULL) return 1;
11     for (i = 0; i < (size_t)size; i++) {
12         p[i] = 0;
13     }
14     printf("%d\n", p[0]);
15     free(p);
16 }
```

Juliet Dynamic

- We adopted Juliet build system to make it suitable for dynamic analysis
- We build each test case in separate binary
- Two versions: with sanitizers and without them
- We measure TP, TN, FP, FN based on Sydr output for version without sanitizers
- Then we verify generated seeds on sanitizers
- Sydr evaluation artifacts are available in Juliet Dynamic repository

Security Predicates Evaluation

CWE	P=N	Textual errors			Sanitizers verification		
		TPR	TNR	ACC	TPR	TNR	ACC
Stack BOF	188	100%	100%	100%	100%	100%	100%
Heap BOF	376	100%	100%	100%	100%	100%	100%
Buffer Underwrite	188	100%	100%	100%	100%	100%	100%
Buffer Overread	188	100%	100%	100%	100%	100%	100%
Buffer Underread	188	100%	100%	100%	100%	100%	100%
Integer Overflow	2580	99.92%	90.89%	95.41%	98.10%	90.89%	94.50%
Integer Underflow	1922	99.90%	91%	95.45%	97.45%	91%	94.22%
Unexpected Sign Ext	752	100%	100%	100%	100%	100%	100%
Signed to Unsigned	752	99.87%	100%	99.93%	99.87%	100%	99.93%
Divide by Zero	564	66.67%	100%	83.33%	66.67%	100%	83.33%
Int Overflow to BOF	188	100%	100%	100%	100%	100%	100%
TOTAL	7886	97.55%	94.83%	96.19%	96.36%	94.83%	95.59%

FreeImage

We found some integer overflow errors during security audit of FreeImage

```
unsigned off_head, off_setup, off_image, i;
...
fseek(ifp, off_setup + 792, SEEK_SET);
```

```
dcraw_common.cpp:15545 - add eax, 0x318 - unsigned integer overflow
dcraw_common.cpp:15545 - call rax - error sink
Found new input "out/int_overflow_10_unsigned"
```

Questions?

No Symbolic Computation

- We just skip some functions to increase performance and reduce overconstrainting
- Dynamic memory: `malloc`, `calloc`, `realloc`, `free`
- Data movement: `strcpy`, `memcpy`, `memmove`, etc.
- Printing omission: `printf`, `std::cout`, `fprintf(stdout)`, etc.

Out-of-bounds Access Strong Precondition

- Sydr conjuncts security predicate with strong precondition to make error most likely cause a crash, i.e. overwrite return address or dereference negative address
- If UNSAT, Sydr falls back to solving the original security predicate

Strong Preconditions and Corner Cases

Strong preconditions:

- Overflowed `*alloc` size argument should be less than original concrete value but not zero
- Overflowed `memcpy` size argument should be greater than original concrete value

Corner cases:

- `SHL`/`SAL` flags do not distinguish integer overflow
- Compiler replaces `sub eax, 1` with `add eax, 0xffffffff`
- Large number arithmetics (`int64_t` on 32-bit)
- Integer promotion and further truncation:

```
char a, b, c;          add edx, esi
c = a + b;             mov BYTE PTR [ebp-0x7], dl
```