Numeric Truncation Security Predicate

Timofey Mezhuev
Ilay Kobrin
Alexey Vishnyakov
Daniil Kuts

December 5, 2023

ISP RAS

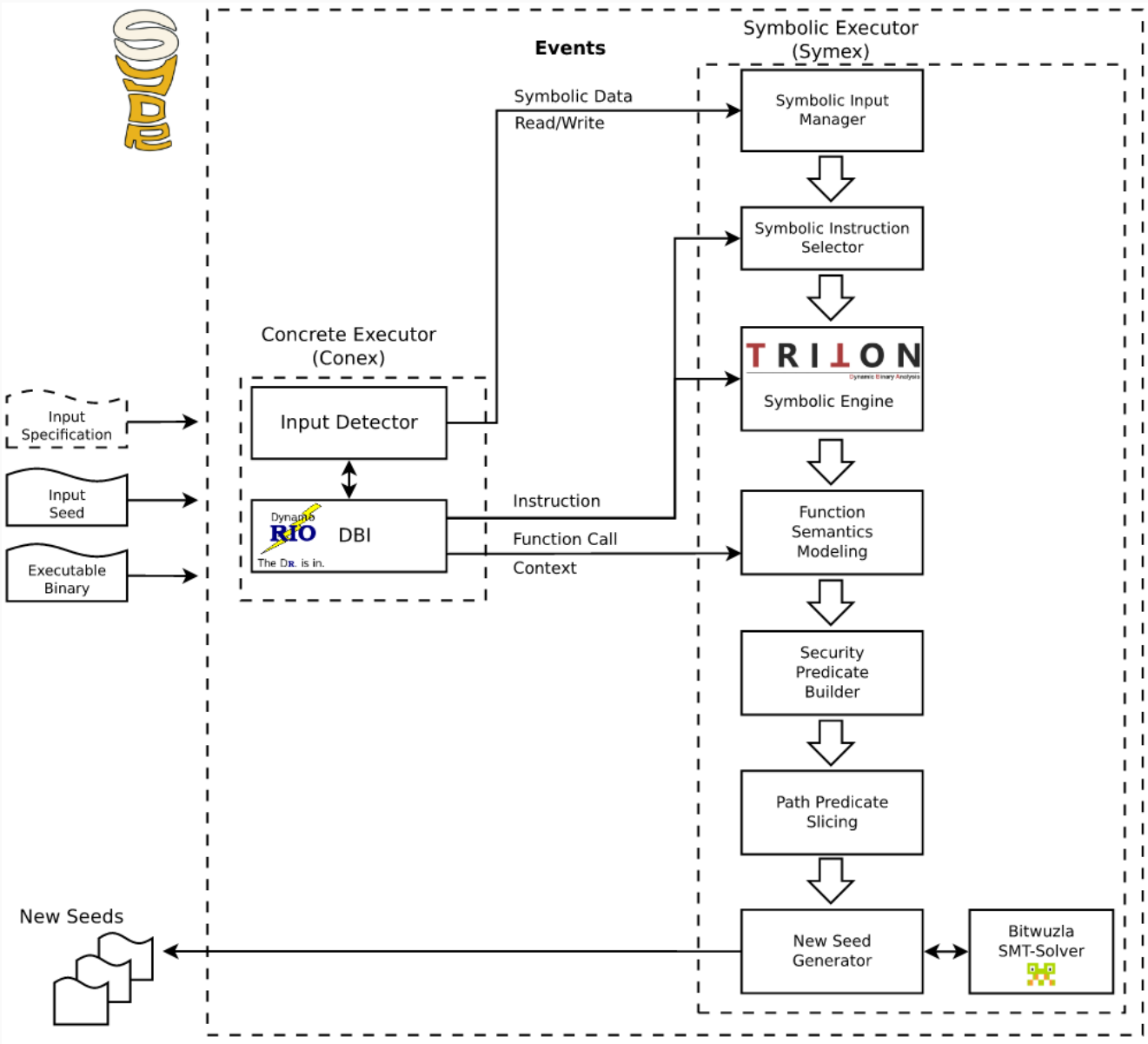
Motivation

- Bugs and vulnerabilities may emerge during the development lifecycle.
- Lots of companies integrate security development lifecycle (SDL) into their workflow processes to detect errors.
- One of the most popular SDL technology is dynamic symbolic execution (DSE).

Dynamic Symbolic Execution

- Dynamic Symbolic Execution is used for analyzing the program at runtime by constructing formulas over symbolic variables.
- These formulas can be analyzed to obtain program execution behavior.
- It is possible to find bugs in binary code with DSE building formulas corresponding to error conditions, which we call security predicates.

Sydr


Symbolic Executor
(Symex)
Events
Symbolic Data
Read/Write
Symbolic Input
Manager
Symbolic Instruction
Selector
TRITON
Symbolic Engine
Function
Semantics
Modeling
Security
Predicate
Builder
Path Predicate
Slicing
New Seed
Generator
Bitwuzla
SMT-Solver
Concrete Executor
(Conex)
Input Detector
DBI
Instruction
Function Call
Context
Input
Specification
Input
Seed
Executable
Binary
New Seeds

Sydr and Security Predicates

Sydr uses Triton as DSE engine, DynamoRIO to instrument binary code instructions and Bitwuzla as SMT-solver.

Dynamic symbolic execution:

- Each input byte is modeled by a free symbolic variable.
- Instruction interpretation produce SMT formulas.
- Path predicate contains taken branch constraints.
- Sydr inverts branches to explore new paths and solves security predicates (integer overflow, null pointer dereference, etc.).
- Sydr sends security predicate to Bitwuzla which generates the input file to reproduce the error in case of successful solution of the constructed formula.

Numeric Truncation

- Numeric truncation error occurs when a value with the bigger type size is converted to the smaller type.
- This error is typical for such programming languages as C/C++, Java, etc.
- Numeric truncation can lead to incorrect program execution or even to some vulnerabilities.
- For example, truncated value of allocated memory size may later cause memory corruptions; or truncated value in loop condition may lead to infinite loop.

Simple example of Numeric Truncation

- 64-bit program
- Input: +00000065536
- Numeric Truncation in line 8
- Output: +00000000000

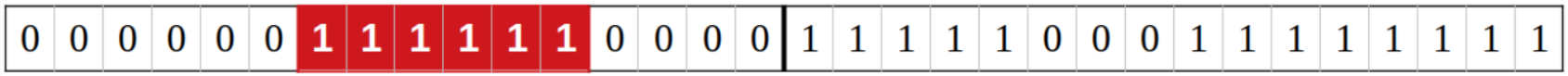
```
1   #include <stdio.h>
2   #include <stdint.h>
3
4   int
5   main() {
6       uint32_t a = 0;
7       scanf("%u", &a);
8       uint16_t b = a;
9       printf("%u\n", b);
10      return 0;
11  }
```

Basic Algorithm

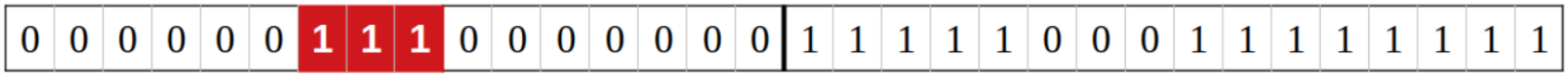
- We analyze every symbolic instruction during the symbolic execution.
- When we meet instructions like `mov`, `movsx`, `movzx`, `cbw`, `cwde`, `cdqe`, we check them for the numeric truncation error by building and solving security predicate.
- For `mov*` instructions: if the initial size of the symbolic value, located in register or memory, is bigger than the source operand size, then we check the security predicate.
- For convert instructions: if the size of the symbolic value located in `rax` register is bigger than the size of the value being extended after `cbw`, `cwde`, `cdqe` instructions execution, then numeric truncation error may occur.

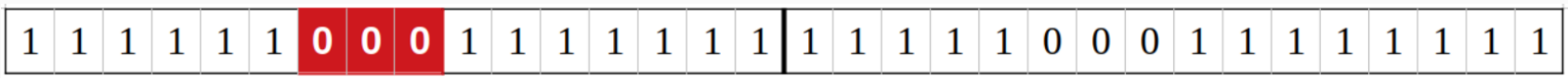
Security Predicate formulas

Unsigned truncation example:



Signed truncation examples:





False signed truncation examples:

Security Predicate formulas

Formula in security predicate:

Unsigned formula:	**Signed formula:**
$not(\phi_{trunc} == bv(0, sz))$	$not((\phi_{trunc} == bv(0, sz)) \vee (\phi_{trunc} == bv(1, sz)))$

ϕ_{trunc} — formula of bits being trucated

$bv(0, sz)$ — bitvector of sz zeros, $bv(1, sz)$ — bitvector of sz ones

- We get the signedness when the value was read with functions like `fscanf, scanf`, etc., that use `strto*` functions inside.
- Otherwise, we get it with the backward slicing algorithm. We find branching instructions like `jl, ja, jb`, etc. to guess signedness.
- If we couldn't get the signedness, we build security predicate with conjunction of signed and unsigned formulas.

False Positive Example

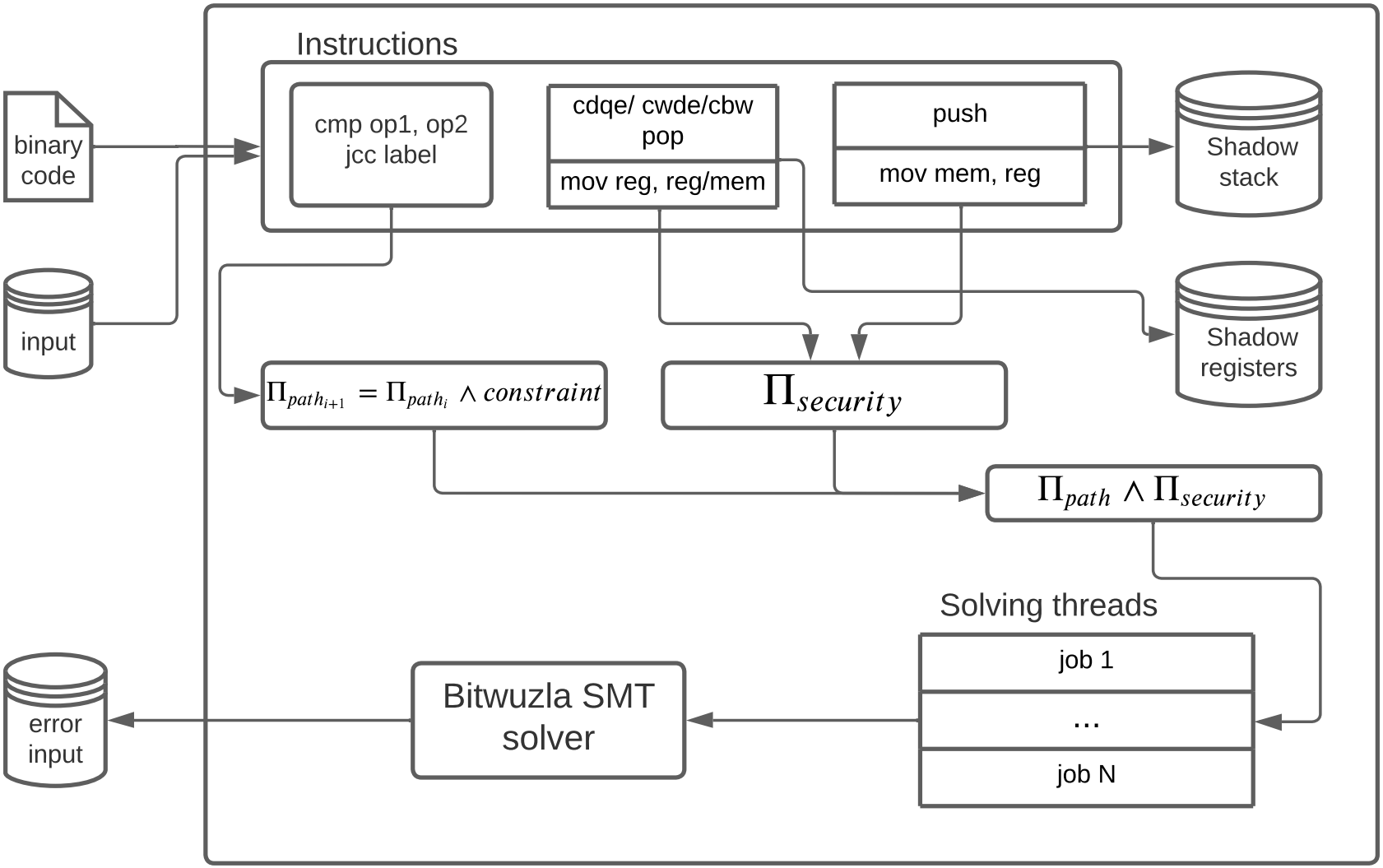
```c
#include <stdio.h>
#include <stdint.h>

void
foo(int16_t a, int8_t b)
{
    printf("%d%d", a, b);
}

int
main()
{
    int32_t i32;
    scanf("%d", &i32);
    foo(i32, i32);
    return 0;
}
```

```
00001255 <main>:
call   10c0 <__isoc99_scanf@plt>
add    esp,0x10
mov    eax,DWORD PTR [ebp-0x10]
movsx  edx,al
mov    eax,DWORD PTR [ebp-0x10]
cwde
push   edx
push   eax
call   120d <foo>
...
0000120d <foo>:
...
mov    edx,DWORD PTR [ebp+0x8]
mov    ecx,DWORD PTR [ebp+0xc]
mov    WORD PTR [ebp-0xc],dx
mov    edx,ecx
mov    BYTE PTR [ebp-0x10],dl
```

Full Scheme (implementation)



Shadow Stack

- To save the size of symbolic memory operands we have Symbolic Shadow Stack. To fill the shadow stack we analyze `mov`, `movsx`, `movzx`, `push` instructions.
- For `mov`, `movsx`, `movzx` instructions:
  1. Check if the destination operand is memory and source operand is a symbolic register.
  2. Check if the source register is in shadow register map.
  3. If so, fill the shadow stack with minimum of source operand size and the size from shadow registers. Otherwise, fill it with the size of source operand.
- For `push` we search the source operand register in shadow register map, and then we fill the current `sp` value address in shadow stack the same way as for `mov`, `movsx`, `movzx` instructions.

Shadow Registers

- To keep up-to-date the size of symbolic register operands we have Symbolic Shadow Registers. To fill the shadow registers we analyze `mov`, `movsx`, `movzx`, `cbw`, `cwde`, `cdqe` and `pop` instructions.
- For `mov`, `movsx`, `movzx` instructions:
  1. Check if destination operand is register and source operand is symbolic.
  2. Check if the source register is in shadow register map (shadow stack).
  3. If so, fill the shadow register with minimum of source operand size and the size from shadow registers (shadow stack). Otherwise, fill it with the size of source operand.
- In case of `pop` instruction we try to get symbolic size from shadow stack for current `sp` value and then save it in shadow registers map for the register operand.

Shadow Registers

- For `cbw`, `cwde`, `cdqe` instructions we update the shadow registers map with the size being extended.
- In case of other instructions we get all the registers written by the instruction and update the shadow registers map with the size of written value. This is necessary to keep up-to-date actual symbolic sizes when the arithmetic operations are performed.

Juliet Dynamic

- In previous works we have adapted Juliet test suite to make it suitable for dynamic analysis. It builds all the tests for specified CWE to binaries in 32-bit and 64-bit modes. Then it runs the tool under test on all the binaries with the sample input data.
- We tested our numeric truncation security predicate on Juliet Dynamic CWE-197 and our approach showed 100% accuracy.

Open Source Trophies

Sydr has found 12 new numeric truncation errors in open-source projects. All of them were reported and fixed.

Project	Detected errors number
nDPI	7
libpcap	2
FreeImage	1
LibTIFF	1
unbound	1

— approved by maintainers

— not approved by maintainers

github.com/ispras/oss-sydr-fuzz/blob/master/TROPHIES.md

nDPI

```
struct diameter_header_t {
  u_int8_t  com_code[3];
  ...
};

typedef enum {
    AC = 271,
    AS = 274,
    CC = 272,
    CE = 257,
    DW = 280,
    DP = 282,
    RA = 258,
    ST = 275
} com_type_t;
...
```

```
u_int16_t com_code =
diameter->com_code[2]
+ (diameter->com_code[1] << 8)
+ (diameter->com_code[0] << 8);

if (com_code == AC ||
    com_code == AS ||
    com_code == CC ||
    com_code == CE ||
    com_code == DW ||
    com_code == DP ||
    com_code == RA ||
    com_code == ST)
return 0;
```

LibTIFF

```
1  static void TIFFReadDirectoryCheckOrder(
2    TIFF *tif, TIFFDirEntry *dir,
3    uint16_t dircount)
4  {
5    static const char module[]
6    = "TIFFReadDirectoryCheckOrder";
7    uint16_t m;
8    uint16_t n;
9    TIFFDirEntry *o;
10   m = 0;
11   for (n = 0, o = dir;
12        n < dircount;
13        n++, o++)
14   {
15     ...
16   }
17 }
```

```
1  if (o->tdir_tag < m)
2  {
3      TIFFWarningExtR(
4      tif, module,
5      "Invalid TIFF directory;"
6      "tags are not sorted in "
7      "ascending order");
8      break;
9  }
10 m = o->tdir_tag + 1;
```

libpcap

```
1         uint16_t size;
2         ...
3         size = tlv->tlv_length;
4         if (size % 4 != 0)
5           size += 4 - size % 4;
6
7         if (size < sizeof(nflog_tlv_t)) {
8           /* Yes. Give up now. */
9           return;
10        }
11
12        if (caplen < size || length < size) {
13          /* No. */
14          return;
15        }
16      ...
```

Questions?

Security Predicate formulas

Unsigned formula:	**Signed formula:**
$not(\phi_{trunc} == bv(0, sz))$	$not((\phi_{trunc} == bv(0, sz)) \vee (\phi_{trunc} == bv(1, sz)))$

$\phi_{trunc} = extract(high - 1, low, \phi_{var})$ — significant bits being truncated from the original value

`high` — original symbolic size, `low` — size of resulting value, ϕ_{var} — symbolic variable formula

$bv(0, sz)$ — bitvector of sz zeros, $bv(1, sz)$ — bitvector of sz ones

$sz = high + low - 1$ — size of ϕ_{trunc} formula

nDPI

```
1  void ndpi_data_add_value(
2      struct ndpi_analyze_struct *s,
3      const u_int64_t value) {
4  if(!s)
5    return;
6  if(s->sum_total == 0)
7    s->min_val = s->max_val = value;
8  else {
9    if(value < s->min_val)
10     s->min_val = value;
11   if(value > s->max_val)
12     s->max_val = value;
13 }
14 s->sum_total += value,
15 s->num_data_entries++;
16 if(s->num_values_array_len) {
17   s->values[s->next_value_insert_index]
18     = value;
19 }
20 ...
21 }
```

github.com/ntop/nDPI/pull/1999

unbound

```
1  int sldns_str2wire_type_buf(const char * str,
2  uint8_t* rd, size_t* len)
3  {
4    uint16_t t = sldns_get_rr_type_by_name(str);
5    ...
6  }
7
8  sldns_rr_type
9  sldns_get_rr_type_by_name(const char *name)
10 {
11   ...
12   if (strlen(name) > 4 && strncasecmp(name,
13   "TYPE", 4) == 0) {
14     return atoi(name + 4);
15   }
16   ...
17 }
```

github.com/NLnetLabs/unbound/pull/909