Symbolic execution for security researchers

Arnau Gàmez i Montolio

EuskalHack Security Congress VI - June 23, 2023 - Donostia

About

Arnau Gàmez i Montolio

Hacker, Reverse Engineer & Mathematician

Occupation

- Senior Expert Engineer, Security @ Activision-Blizzard
- Founder, Researcher & Trainer @ Fura Labs
- PhD @ City, University of London

Social

- Tweets: @arnaugamez, @furalabs
- Toots: {@arnaugamez, @furalabs}@infosec.exchange

Preliminaries

Expectations

- Aims to be pretty introductory
- Demystify symbolic execution for a non-specialized audience
- Focus on understanding ideas rather than specific tooling
    - Apply it to your own areas of interest
    - Make it easy to use *any* tools (and understand what you are doing)
    - Make it easy to even contribute to, or write your own (open source) tools

Calculator

Concrete calculations

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = 4 - 2 \cdot 3 = -2$$

Calculator

Concrete calculations

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = 4 - 2 \cdot 3 = -2$$

Computer Algebra System (CAS)

Symbolic calculations and expression manipulation

$$\begin{vmatrix} 1 & 2 \\ a & 4 \end{vmatrix} = 4 - 2a = 2(2 - a)$$

Intermediate representation / language (IR/IL)

Language of an abstract machine designed to aid in the analysis of computer programs:

- Compilation: common ground for architecture independent processing
- Decompilation (binary analysis): lifting from ASM to canonical *higher level* representation
- Transpiling: source to source compilation

What is symbolic execution?

Roughly speaking, just a **computer algebra system** for:

- Programming languages: C, C++, Java, Rust...
- Assembly languages: x86, x86-64, ARM64, MIPS, RISC-V...
- Intermediate languages: LLVM-IR, SMT-LIB, r2 ESIL, IDA Microcode, $YOUR_OWN...

More specifically, symbolic execution is a **program analysis technique**:

- Represent inputs as *symbolic* variables instead of *concrete* values (normal execution or emulation)
- Derive constraints that encode control-flow and data-flow with respect to these symbolic variables

Use these constraints to reason about and extract information from the program

```
int foo(int x, int y) {
    x = y - 3*x;
    if (x < y) {
        return 2*x - x^y;
    }
    else {
        return 3*y + x|y;
    }
}
```

```c
int foo(int x, int y) {
    x = y - 3*x;
    if (x < y) {
        return 2*x - x^y;
    }
    else {
        return 3*y + x|y;
    }
}
```

Concrete execution (or emulation)

$$x = 1337, y = 7331$$

$$x = 7331 - 3 \times 1337 = 3320$$

$$(3320 < 7331)$$

$$2 \times 3320 - 1337 \oplus 7331 = 2068$$

$$\hookrightarrow 2068$$

```c
int foo(int x, int y) {
    x = y - 3*x;
    if (x < y) {
        return 2*x - x^y;
    }
    else {
        return 3*y + x|y;
    }
}
```

Concrete execution (or emulation)

$$x = 1337, y = 7331$$
$$x = 7331 - 3 \times 1337 = 3320$$

$$(3320 < 7331)$$
$$2 \times 3320 - 1337 \oplus 7331 = 2068$$
$$\hookrightarrow 2068$$

Symbolic execution

$$x = \mathbf{x}, y = \mathbf{y}$$
$$\mathbf{x} = \mathbf{y} - 3\mathbf{x}$$

```
int foo(int x, int y) {
    x = y - 3*x;
    if (x < y) {
        return 2*x - x^y;
    }
    else {
        return 3*y + x|y;
    }
}
```

Concrete execution (or emulation)

$$x = 1337, y = 7331$$

$$x = 7331 - 3 \times 1337 = 3320$$

$$(3320 < 7331)$$

$$2 \times 3320 - 1337 \oplus 7331 = 2068$$

$$\hookrightarrow 2068$$

Symbolic execution

$$x = \mathbf{x}, y = \mathbf{y}$$

$$\mathbf{x} = \mathbf{y} - 3\mathbf{x}$$

$$(\mathbf{y} - 3\mathbf{x} < \mathbf{y})$$

$$\hookrightarrow 2(\mathbf{y} - 3\mathbf{x}) - (\mathbf{y} - 3\mathbf{x}) \oplus \mathbf{y}$$

```c
int foo(int x, int y) {
    x = y - 3*x;
    if (x < y) {
        return 2*x - x^y;
    }
    else {
        return 3*y + x|y;
    }
}
```

Concrete execution (or emulation)

$$x = 1337, y = 7331$$

$$x = 7331 - 3 \times 1337 = 3320$$

$$(3320 < 7331)$$

$$2 \times 3320 - 1337 \oplus 7331 = 2068$$

$$\hookrightarrow 2068$$

Symbolic execution

$$x = \mathbf{x}, y = \mathbf{y}$$

$$\mathbf{x} = \mathbf{y} - 3\mathbf{x}$$

$$(\mathbf{y} - 3\mathbf{x} < \mathbf{y})$$

$$\hookrightarrow 2(\mathbf{y} - 3\mathbf{x}) - (\mathbf{y} - 3\mathbf{x}) \oplus \mathbf{y}$$

$$(\mathbf{y} - 3\mathbf{x} \geq \mathbf{y})$$

$$\hookrightarrow 3(\mathbf{y} - 3\mathbf{x}) - (\mathbf{y} - 3\mathbf{x}) \vee \mathbf{y}$$

But how does it *actually* work?

1. Define two data structures:
    - **path_constraint**: conditions required to reach current instruction
    - **state_map**: symbolic mapping for the variables (registers, memory locations)
2. Extract the semantics of each statement (instruction)
3. Update these two data structures to account for the effects of the *executed* statement (instruction)
4. If there is control-flow branching, *fork* these structures to keep track of different execution paths

The **state_map** represents *data-flow* updates, i.e. the (computational) process though which a variable ends up holding a certain value at a given point in the program execution.

The **state_map** represents *data-flow* updates, i.e. the (computational) process though which a variable ends up holding a certain value at a given point in the program execution.

The **path_constraint** represents *control-flow* tracking, i.e. the set of constraints (conditions) on the variables that need to be satisfied for the execution to reach a given point in the program.

Visual example

```
_start:
    mov rax, 123  <=0=
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> rax
                 rbx -> rbx
                 rdi -> rdi
                 rsi -> rsi
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi  <=0=
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
```

```
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> 123
                 rbx -> rbx
                 rdi -> rdi
                 rsi -> rsi
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi   <=0=
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> (123 + rsi)
                 rbx -> rbx
                 rdi -> rdi
                 rsi -> rsi
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2      <=0=
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> ((123 + rsi) ^ rdi)
                 rbx -> rbx
                 rdi -> rdi
                 rsi -> rsi
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx   <=0=
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> ((123 + rsi) ^ rdi)
                 rbx -> 2
                 rdi -> rdi
                 rsi -> rsi
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3     <=0=
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi

    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> (((123 + rsi) ^ rdi) + 2)
                 rbx -> 2
                 rdi -> rdi
                 rsi -> rsi
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax  <=0=
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> (((123 + rsi) ^ rdi) + 2)
                 rbx -> 2
                 rdi -> 3
                 rsi -> rsi
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx  <=0=
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> (((123 + rsi) ^ rdi) + 2)
                 rbx -> 2
                 rdi -> 3
                 rsi -> (((123 + rsi) ^ rdi) + 2)
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi  <=0=
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> ((((123 + rsi) ^ rdi) + 2) + 2)
                 rbx -> 2
                 rdi -> 3
                 rsi -> (((123 + rsi) ^ rdi) + 2)
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7     <=0=
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> (((((123 + rsi) ^ rdi) + 2) + 2) ^ 3)
                 rbx -> 2
                 rdi -> 3
                 rsi -> (((123 + rsi) ^ rdi) + 2)
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx  <=0=
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> (((((123 + rsi) ^ rdi) + 2) + 2) ^ 3)
                 rbx -> 7
                 rdi -> 3
                 rsi -> (((123 + rsi) ^ rdi) + 2)
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336 <=0=
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> ((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7)
                 rbx -> 7
                 rdi -> 3
                 rsi -> (((123 + rsi) ^ rdi) + 2)
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi  <=0=
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> ((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7)
                 rbx -> 7
                 rdi -> 1336
                 rsi -> (((123 + rsi) ^ rdi) + 2)
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337 <=0=
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> (((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7) + 1336)
                 rbx -> 7
                 rdi -> 1336
                 rsi -> (((123 + rsi) ^ rdi) + 2)
                 zf  -> zf
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad        <=0=

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```

```
path_constraint  true
state_map        rax -> (((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7) + 1336)
                 rbx -> 7
                 rdi -> 1336
                 rsi -> (((123 + rsi) ^ rdi) + 2)
                 zf  -> (((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7) + 1336)
                        == 1337 ? 1 : 0
```

```
_start:
    mov rax, 123
    add rax, rsi
    xor rax, rdi
    mov rbx, 2
    add rax, rbx
    mov rdi, 3
    mov rsi, rax
    add rax, rbx
    xor rax, rdi
    mov rbx, 7
    and rax, rbx
    mov rdi, 1336
    add rax, rdi
    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi   <=1=
    jmp exit

bad:
    mov rdi, 1     <=2=

exit:
    mov rax, 60
    syscall
```

```
path_constraint  (((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7) + 1336) == 1337
state_map        ...
                 zf  -> 1

path_constraint  (((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7) + 1336) != 1337
state_map        ...
                 zf  -> 0
```

How do we *reason* about this information?

How do we *reason* about this information?

With an SMT solver

How do we *reason* about this information?

With an SMT solver

Mostly

SMT solver

SMT solver

Satisfiability Modulo Theories

- **S**atisfiability (SAT): determine if a (boolean) formula can be satisfied (can be true)
- **M**odulo: take into account (not only boolean formulas but also)...
- **T**heories: ...integer numbers, real numbers, floating point, **bit vectors**, and more

SMT solver

Satisfiability Modulo Theories

- **S**atisfiability (SAT): determine if a (boolean) formula can be satisfied (can be true)
- **M**odulo: take into account (not only boolean formulas but also)...
- **T**heories: ...integer numbers, real numbers, floating point, **bit vectors**, and more

From a very practical standpoint: a *magic black-box* that can only answer a very simple question.

Question

Given some variables of some type, and some constraints on these variables:

- Is there any variable assignment that makes the set of constraints satisfiable, i.e. such that (all) the constraints hold true?

Question

Given some variables of some type, and some constraints on these variables:

- Is there any variable assignment that makes the set of constraints satisfiable, i.e. such that (all) the constraints hold true?

Outcomes

- SAT: there is a variable assignment that makes all the constraints hold true.
  - It will actually find a model, which is a particular solution (a concrete variable assignment)
- UNSAT: there is NO variable assignment that makes all the constraints hold true.
- UNKNOWN: unable to answer the question (usually due to a time-out)

Symbolic execution + SMT solver

Symbolic execution + SMT solver

Some basic ideas

Control-flow analysis

Control-flow analysis

1. The symbolic execution engine is used to extract the formulae (constraints) for a given path branching to happen: check its `path_constraint`

Control-flow analysis

1. The symbolic execution engine is used to extract the formulae (constraints) for a given path branching to happen: check its `path_constraint`

2. The constraints are fed into the SMT solver

Control-flow analysis

1. The symbolic execution engine is used to extract the formulae (constraints) for a given path branching to happen: check its `path_constraint`

2. The constraints are fed into the SMT solver

3. The SMT solver can prove the feasibility of the constraints, meaning the path is reachable
    - If it is, retrieve a model for it, i.e. input values that will make the program execution to reach it
    - If it is not, we have detected an obfuscating opaque predicate and can ignore/patch it away

Example

```
path_constraint  (((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7) + 1336) == 1337
```

```
path_constraint  (((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7) + 1336) == 1337
```

Given 64-bit variables `rdi` and `rsi`:

- Is there any variable assignment (for `rdi` and `rsi`) that makes the `path_constraint` satisfiable?

```python
import z3

rdi, rsi = z3.BitVecs('rdi rsi', 64)
path_constraint = (((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7) + 1336) == 1337

solver = z3.Solver()
solver.add(path_constraint)

if solver.check() == z3.sat:
    print(solver.model())
```

```python
import z3

rdi, rsi = z3.BitVecs('rdi rsi', 64)
path_constraint = (((((((123 + rsi) ^ rdi) + 2) + 2) ^ 3) & 7) + 1336) == 1337

solver = z3.Solver()
solver.add(path_constraint)

if solver.check() == z3.sat:
    print(solver.model())
```

```
[rdi = 2, rsi = 1]
```

Data-flow analysis

Data-flow analysis

- Embed *compiler optimization* techniques into the `state_map` population process:
  - Constant propagation: by construction
  - Constant folding: evaluate intermediate expressions on constant values
  - Reaching definitions: calculate at a given point the set of definitions that reach it
  - Liveness analysis: calculate at a given point the *live* variables (may be read before updated)

1. The symbolic execution engine is used to extract the formula of the return value of a function with respect to its inputs parameters: check its value in the `state_map`

1. The symbolic execution engine is used to extract the formula of the return value of a function with respect to its inputs parameters: check its value in the `state_map`

2. The formula is fed into the SMT solver

1. The symbolic execution engine is used to extract the formula of the return value of a function with respect to its inputs parameters: check its value in the `state_map`

2. The formula is fed into the SMT solver

3. The SMT can:
    - Attempt to simplify the formula to get a nicer representation
    - Craft inputs value that will make the formula evaluate to a desired output (i.e. inputs that will make the function return a desired value)

Tooling

Tooling

Welcome to the jungle

Implementation technology

- **Interpreter based**: Miasm, Triton, Angr, Maat, radius2
- **Instrumentation based**: QSYM
- **Compiler based**: KLEE, SymCC, SymQEMU

Implementation technology

- **Interpreter based**: Miasm, Triton, Angr, Maat, radius2
- **Instrumentation based**: QSYM
- **Compiler based**: KLEE, SymCC, SymQEMU

Target

- **Binary**: Miasm, Triton, Angr, Maat, radius2, QSYM, SymQEMU
- **Source code**: KLEE, SymCC

Implementation technology

- **Interpreter based**: Miasm, Triton, Angr, Maat, radius2
- **Instrumentation based**: QSYM
- **Compiler based**: KLEE, SymCC, SymQEMU

Target

- **Binary**: Miasm, Triton, Angr, Maat, radius2, QSYM, SymQEMU
- **Source code**: KLEE, SymCC

Focus

- **Analysis**: Miasm, Triton, Maat
- **Automagic**: Angr, radius2
- **Test generation**: QSYM, KLEE, SymCC, SymQEMU

Practical applications

Practical applications

An appetizer

Analysis of complex code

Detect (and patch) opaque predicates

Opaque predicates

A conditional statement P whose truth value is known a priori.

Opaque predicates

A conditional statement P whose truth value is known a priori.

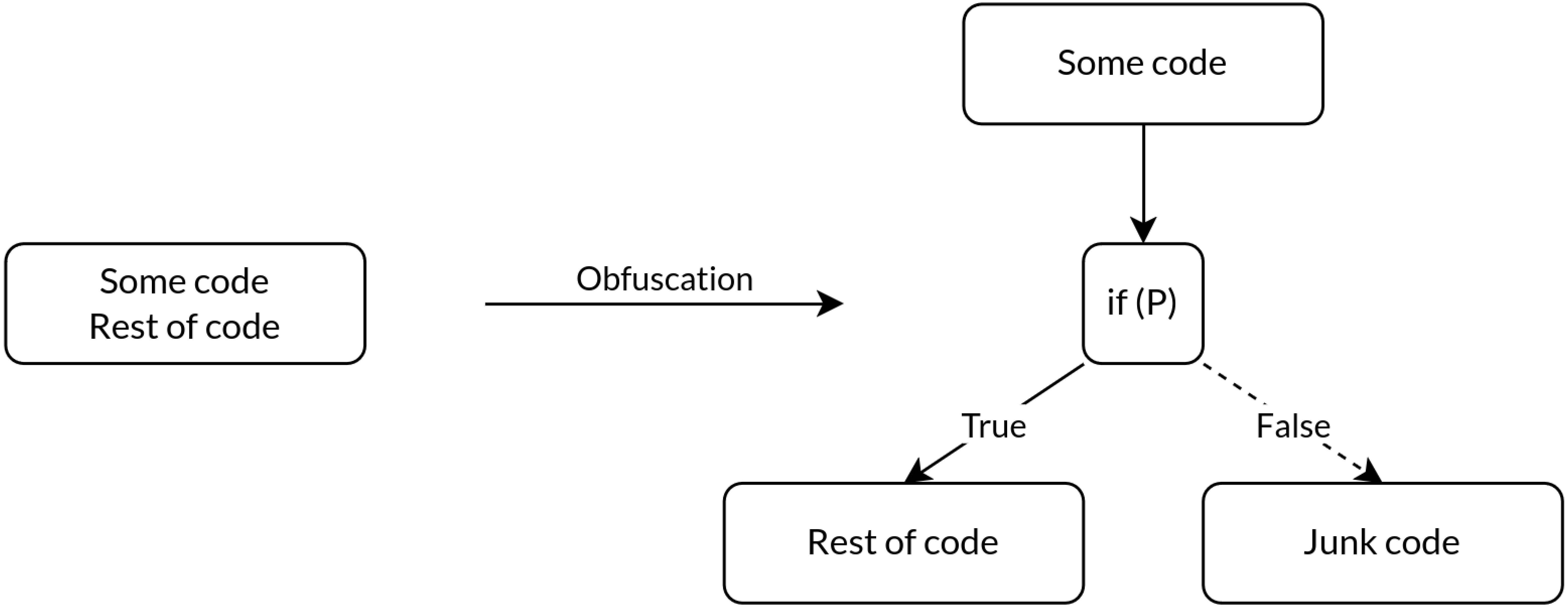
$x^2 \geq 0$ is always true.

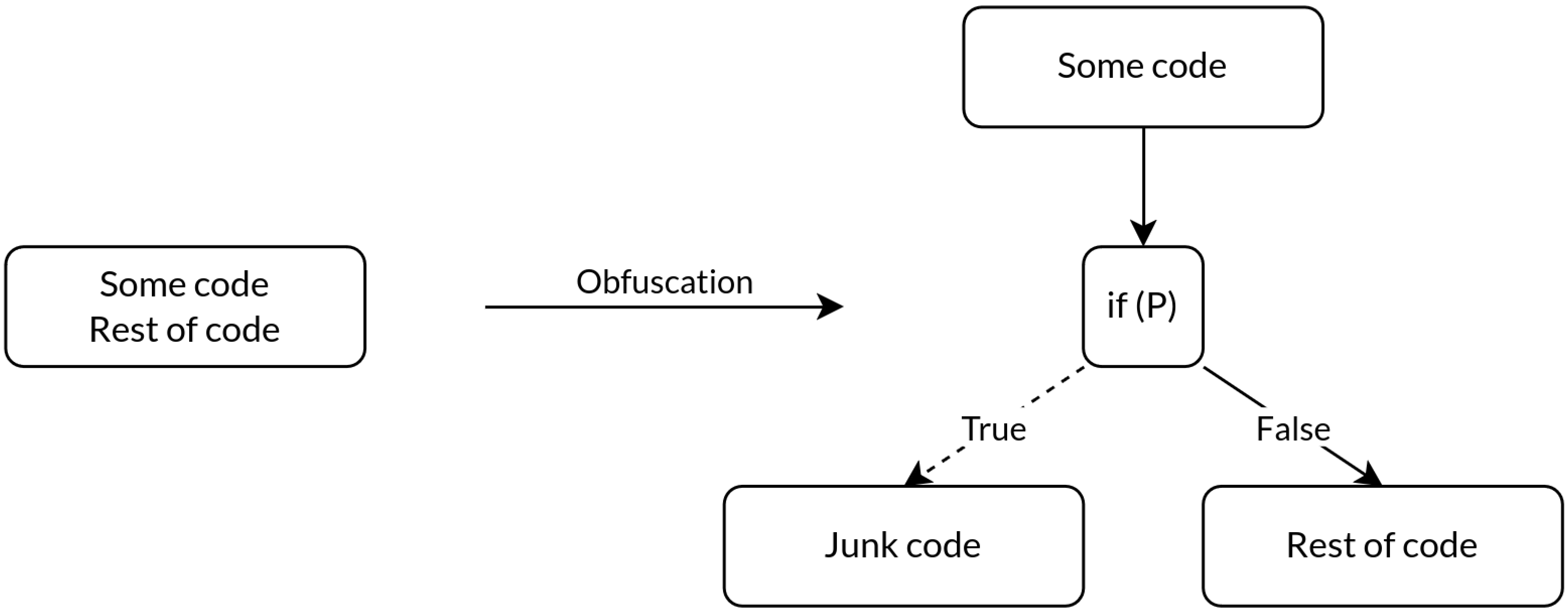
Opaque predicates

A conditional statement P whose truth value is known a priori.

$x^2 \geq 0$ is always true.

$7y^2 - 1 = x^2$ is always false.


Some code
Rest of code
Obfuscation
Some code
if (P)
True
False
Rest of code
Junk code


Some code
Rest of code
Obfuscation
Some code
if (P)
True
False
Junk code
Rest of code

Detect (and patch) opaque predicates

- Symbolically execute a basic block
- Extract the branching constraints
- Check if the constraints are either always true (or false)
- Patch it to continue execution at the only possible branch and remove `NOP` the unreachable branch

Example

Example

XTunnel @ APT28: ac3e087e43be67bdc674747c665b46c2

Example

XTunnel @ APT28: ac3e087e43be67bdc674747c665b46c2

Based on: *https://github.com/mrphrazer/r2con2020_deobfuscation/blob/master/remove_opaque.py* by Tim Blazytko (aka mrphrazer)

```python
# Define function start address and construct asmcfg and ircfg
f_addr = 0x491AA0
asmcfg = dis_engine.dis_multiblock(f_addr)
lifter = machine.lifter_model_call(dis_engine.loc_db)
ircfg = lifter.new_ircfg_from_asmcfg(asmcfg)
```

```python
# Check whether the expr path constraint is compatible with target path constraint
def cannot_branch(expr, target):
    solver = Solver()
    translator = TranslatorZ3() # convert miasm ir into z3

    exp1 = translator.from_expr(expr)
    exp2 = translator.from_expr(target)

    solver.add(exp1 == exp2)
    return solver.check() == unsat
```

```python
# Load the file as raw bytes
xtunnel_bytes = bytearray(open(xtunnel, 'rb').read())
for bb in asmcfg.blocks:
    # Extract address of current basic block
    bb_addr = bb.lines[0].offset

    # Initialize the symbolic execution engine
    symex_engine = SymbolicExecutionEngine(lifter)

    # Execute basic block
    expr = symex_engine.run_block_at(ircfg, bb_addr)
```

```python
    # Check if the basic block branches (conditional expression)
    if expr.is_cond():
        # Check if it CANNOT branch to the TRUE branch
        if cannot_branch(expr, expr.src1):
            # Get the virtual offset of the jump
            jump_inst              = bb.lines[-1]
            jump_virtual_offset    = jump_inst.offset

            # Get the initial and end file offsets for the jump basic block
            jump_file_offset_init = container.bin_stream.bin.virt2off(
                jump_virtual_offset
            )
            jump_file_offset_end  = jump_file_offset_init + len(jump_inst.b)

            # Patch with NOPs
            for byte in range(jump_file_offset_init, jump_file_offset_end):
                xtunnel_bytes[byte] = 0x90 # NOP

open("XTunnel_patched.bin", 'wb').write(xtunnel_bytes)
```

Fuzzing

Increase code coverage

Code coverage

Measure of the degree to which the code of a program is executed when a set of inputs is run.

- Subroutines called
- Statements executed

Higher code coverage → higher chance of hitting *interesting* (vulnerable) code

Increase code coverage

- Start fuzzing your target with an initial seed/corpus
- Leverage symbolic execution:
    1. Check current inputs
    2. Generate inputs that trigger non-explored paths
    3. Feed these new inputs into the fuzzer
    4. Repeat

Example

Example

Trigger a hard-to-reach division by zero

Example

Trigger a hard-to-reach division by zero

Based on: *Fuzzing combined with symbolic execution: a demonstration on SymCC and AFL* by AdaLogics

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int stuff(char *data, long fsize) {
    for (size_t i = 0; i < fsize; i++) {
        if (data[i] == ('F' ^ i)):
            return i+1;
    }

    if (*(int*)data != 0xfa1afe1):
        return 0;

    return (int)(0x1337/(fsize - 10)); // <== TRIGGER DIV BY 0 HERE
}
```

I made a (dumb) SymCC fork (SymCC++) to make it work with AFL++

https://github.com/arnaugamez/symccpp

Demo time

Limitations

Limitations

And some ideas to overcome them

- Path explosion: the number of control-flow paths grows exponentially ($\rightarrow \infty$ for unbounded loops)
    - Manual location of interesting code
    - Concolic (**con**crete + symb**olic**) execution

- Path explosion: the number of control-flow paths grows exponentially ($\rightarrow \infty$ for unbounded loops)
    - Manual location of interesting code
    - Concolic (**con**crete + symb**olic**) execution

- Support for syscalls, standard C library functions, etc.:
    - Same as with any emulator: *hook 'em all*

- Path explosion: the number of control-flow paths grows exponentially ($\rightarrow \infty$ for unbounded loops)
    - Manual location of interesting code
    - Concolic (**con**crete + symb**olic**) execution

- Support for syscalls, standard C library functions, etc.:
    - Same as with any emulator: *hook 'em all*

- Limits of SMT solvers (e.g. due to high algebraic complexity through MBA transformations):
    - Program synthesis
    - Maths™
    - Imagination

Training

An analytical apporach to modern binary deobfuscation

A curated training that teaches you to build, analyze and defeat obfuscated code

- Public / Private
- In-person / Remote
- 4 days (flexible)
- Details: https://furalabs.com/trainings

Upcoming

- August 05-08, 2023 @ RingZer0 - Las Vegas (USA)
- August 21-24, 2023 @ HITB - Remote

Thank you

Q&A