

# Hands-on binary (de)obfuscation

**Arnau Gàmez i Montolio**

Ringzer0 BOOTSTRAP25 - March 22, 2025 - Austin (TX)

# Agenda

- **Introduction:** Software protection landscape
- **Preliminary:** SMT
- **Part I:** Mixed Boolean-Arithmetic (MBA) obfuscation
- **Part II:** Analysis - Symbolic execution
- **Part III:** Analysis - Program synthesis

# About

Arnau Gàmez i Montolio

Hacker, Reverse Engineer & Mathematician

## Occupation

- Senior Expert Security Engineer @ Activision
- PhD @ City St George's, University of London
- Founder, Researcher & Trainer @ Fura Labs

# Introduction

## Software protection landscape

## Context

Protection against Man-At-The-End (MATE) attacks.

The attacker has an instance of the program and controls the environment it is executed in.

## Protection against end users

### Technical

- Obfuscation
- Cryptography
- Server-side execution
- Trusted execution environment (TEE)
- Device attestation
- ...

### Legal

- Lawyers
- Luck
  - Jurisdiction
  - Adversary's strength
- Patience
- ...

# Obfuscation

Transform a (part of a) program  $P$  into a functionally equivalent (part of a) program  $P'$  which is harder to analyze and extract information from than  $P$ .

$$P \longrightarrow \boxed{\text{Obfuscation}} \longrightarrow P'$$

## Motivation

~~Prevent~~ Prevent complicate reverse engineering.

## Presence

### Commercial software

- Intellectual property
- Digital Rights Management (DRM)
- (Anti-)cheating

### Malware

- Avoid automatic signature detection
- Slow down analysis → time → money



## Methodology

Semantics-preserving transformations to data-flow procedures and control-flow structures.

### At different abstraction levels

- Source code
- Intermediate representation
- Assembly listing
- Compiled binary

### At different target units

- Whole program
- Function
- Basic block
- Instruction

**Remark:** Several *weak* techniques can be combined to create *hard* obfuscation transformations.

# Deobfuscation

Transform an obfuscated (part of a) program  $P'$  into a (part of a) program  $P''$  which is easier to analyze and extract information from than  $P'$ .

$$P'' \longleftarrow \boxed{\text{Deobfuscation}} \longleftarrow P'$$

Ideally  $P'' \approx P$ , but this is rarely the case:

- Lack of access to original program  $P$ .
- Interest in specific parts rather than whole program.
- Interest in understanding rather than rebuilding.

# Preliminary

## SMT

## Satisfiability Modulo Theories (SMT)

- **Satisfiability (SAT)**: determine if a (boolean) formula can be satisfied (can be true).
- **Modulo**: take into account (not only boolean formulas but also)...
- **Theories**: ...integer numbers, real numbers, floating point, **bit vectors**, and more.

## SMT solver

In practice: a *magic black-box* that can only answer a very simple question.

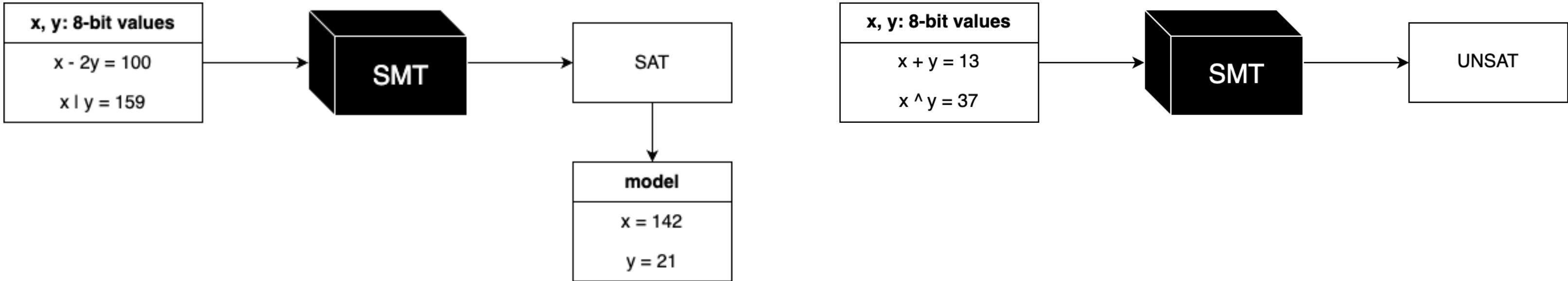
## Question

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?

## Outcomes

- **SAT**: there is a variable assignment that makes all the constraints hold true.
  - It actually finds 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).



# Program analysis with an SMT solver

- Check semantic equivalence
- Simplification engine
- Solve complex constraints
- Input crafting
- Model counting

# Limitations

- Resource exhaustion.
- Since SAT is NP-complete, SMT problems are *at least* NP-complete.
- Expression complexity.
  - Due to underlying semantic complexity (e.g. any decent cryptosystem).
  - Due to deliberate obfuscation (e.g. complex algebraic transformations).



# Part I

## Mixed Boolean-Arithmetic (MBA) obfuscation

# MBA expressions

Algebraic expressions composed of integer arithmetic operators ( $+$ ,  $-$ ,  $\times$ ) and bitwise operators ( $\wedge$ ,  $\vee$ ,  $\oplus$ ,  $\neg$ ).

Operation	Math	Code
AND	$\wedge$	<code>&amp;</code>
OR	$\vee$	<code> </code>
XOR	$\oplus$	<code>^</code>
NOT	$\neg$	<code>~</code>

**Note:** I will use interchangeably the terms *boolean*, *bitwise* and *logic* operators.

## Linear MBA expressions

$$(x \oplus y) + 2 \times (x \wedge y)$$

## Polynomial MBA expressions

$$43(x \wedge y \vee z)^2((x \oplus y) \wedge z \vee t) + 2x + 123(x \vee y)zt^2$$

## Obfuscate expressions

Given an MBA expression  $E_1$ , generate an expression  $E_2$  that is:

- Semantically equivalent to  $E_1$ .
- Syntactically more complex than  $E_1$ .

For that, we have *rewrite rules* and *insertion of identities*.

## Rewrite rules

Replace an expression with an equivalent (more complex) one.

$$x + y \rightarrow (x \oplus y) + 2 \times (x \wedge y)$$

**Note:** They can be applied iteratively (due to composability of polynomial MBA expressions).

$$x + y \rightarrow (x \oplus y) + 2(x \wedge y)$$



$$x' + y'$$

$$x' = (x \oplus y)$$

$$y' = 2(x \wedge y)$$

$$x' + y' \rightarrow (x' \oplus y') + 2(x' \wedge y') \equiv ((x \oplus y) \oplus 2(x \wedge y)) + 2((x \oplus y) \wedge 2(x \wedge y))$$

$$\circlearrowleft$$

$$x' + y'$$

$$x' = ((x \oplus y) \oplus 2(x \wedge y))$$

$$y' = 2((x \oplus y) \wedge 2(x \wedge y))$$

## Insertion of identities

Wrap an expression with a pair of invertible mappings.

$$e = (x \oplus y) + 2 \times (x \wedge y) \quad f : x \mapsto 39x + 23 \quad f^{-1} : x \mapsto 151x + 111$$

$$f^{-1}(f(e)) = 151 \times (39 \times ((x \oplus y) + 2 \times (x \wedge y)) + 23) + 111$$

**Note:** In general, affine functions (or permutation polynomials).



# Obfuscate constants

Replace a constant by a computational process (expression) on a given number of variables that will always evaluate to the target constant at runtime.

## Opaque constants

- $K$  constant
- $P, Q$  inverse permutation polynomials
- $E$  non-trivially equal to zero MBA expression

Conceal constant:  $K \equiv P(E + Q(K))$

Proof:

$$P(E + Q(K)) = P(0 + Q(K)) = P(Q(K)) = K$$

$$K = 123$$

$$P(X) = 97X + 248X^2$$

$$Q(X) = 161X + 136X^2$$

$$E(x, y) = x - y + 2(\neg x \wedge y) - (x \oplus y)$$

$$\begin{aligned} P(E + Q(K)) = & 195 + 97x + 159y + 194\neg(x \vee \neg y) + 159(x \oplus y) \\ & + (163 + x + 255y + 2\neg(x \vee \neg y) + 255(x \oplus y)) \\ & \times (232 + 248x + 8y + 240\neg(x \vee \neg y) + 8(x \oplus y)) \end{aligned}$$

## Fact

State-of-the-art software protection leverages MBA transformations to obfuscate code.

## Why?

Combinations of operators from these different fields *do not interact well together*.

- No general rules (distributivity, factorization...) or theory.
- Computer algebra systems do not support bitwise operators with symbolic variables.

SMT solvers support for mixing operators (*bit vector theory*).

- Reasonably good at proving semantic equivalence.
  - Easily thwarted with deliberate MBA transformations, though.
- Pretty bad at simplification for general MBA expressions.

# Hands-on

# Part II

## Analysis - Symbolic execution



## Calculator

Concrete calculations.

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

## Computer Algebra System (CAS)

Symbolic calculation and manipulation.

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

# 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**:

1. Represent inputs as *symbolic* variables instead of *concrete* values.
2. Derive constraints that encode control-flow and data-flow with respect to them.
3. Use these constraints to reason about and extract information from the program.

## But how does it *actually* work?

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

The **state\_map** represents *data-flow* updates, i.e. the (computational) process through 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

```

```

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

```

```

      cmp rax, 1337
      jnz bad

good:
      xor rdi, rdi
      jmp exit

bad:
      mov rdi, 1

exit:
      mov rax, 60
      syscall

```

```
_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
```

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

```
        cmp rax, 1337
        jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall
```



```

_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

```

```

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

```

```

    cmp rax, 1337
    jnz bad

good:
    xor rdi, rdi
    jmp exit

bad:
    mov rdi, 1

exit:
    mov rax, 60
    syscall

```

```

_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
    jmp exit

bad:
    mov rdi, 1

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?**

**With an SMT solver**

Mostly

# Data-flow analysis



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).

## Compiler optimization techniques

Embedded 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).

# Control-flow analysis

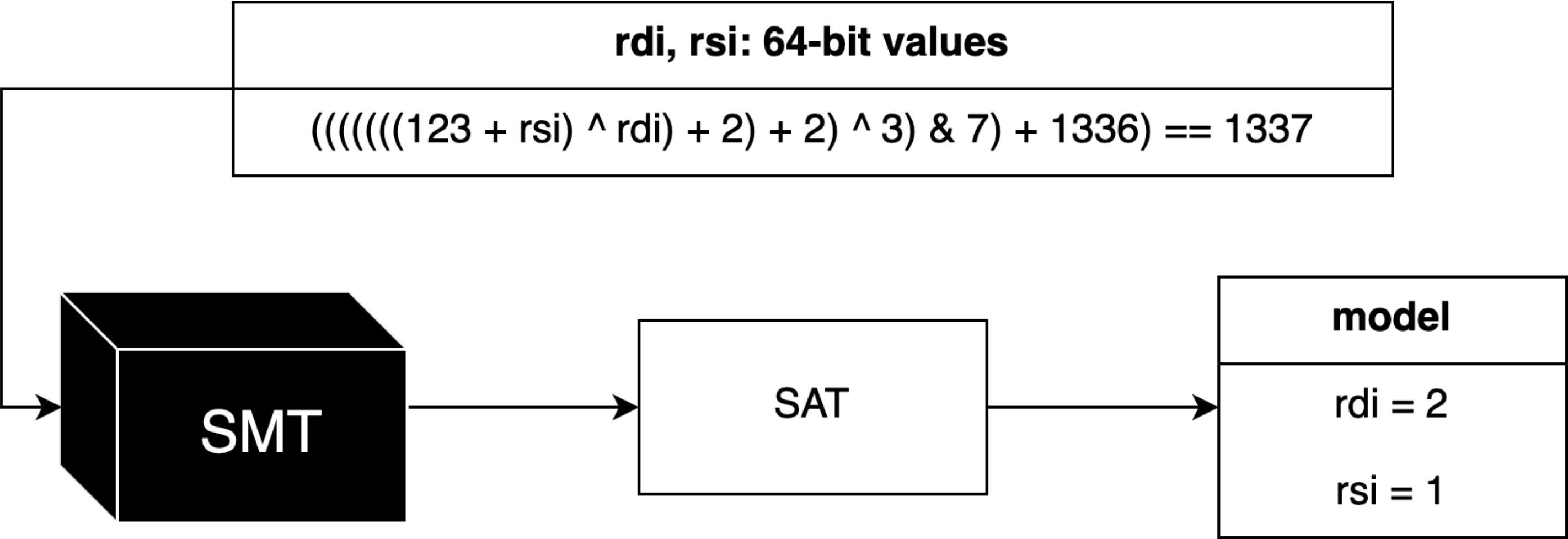
1. The symbolic execution engine is used to extract the formulae (constraints) for a given path branching to happen: check its `path_constraint`.
2. The constraints are fed into the SMT solver.
3. The SMT solver can prove the feasibility of the constraints (path reachability):
  - SAT: retrieve a model for it, i.e. concrete input values that reach the path
  - UNSAT: an obfuscating opaque predicate is detected -> ignore/patch it away

## Example

```
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?



```
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]
```



# Tooling

Welcome to the jungle

## 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, (Angr?)
- **Automagic:** Angr, radius2
- **Test generation:** QSYM, KLEE, SymCC, SymQEMU

# Limitations

**And some ideas to overcome them**

## Path explosion

The number of control-flow paths grows exponentially ( $\rightarrow \infty$  for unbounded loops).

### Approach

- Manual location of interesting code.
- Concolic (**con**crete + **sym**bo**lic**) execution.

## Environment support

Syscalls, WinAPI, standard C library...

## Approach

- Same as with any emulator: hooks and stubs.

## Limits of SMT solvers

Complex expressions (MBA alternation, high algebraic degree...).

### Approach

- Program synthesis
- Math<sup>TM</sup>
- Imagination

# Hands-on

# Part III

## Analysis - Program synthesis



## Motivating example

Consider the following obfuscated expression:

$$f(x, y, z) = (((x \oplus y) + ((x \wedge y) \times 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \times 2)) \wedge z)$$

Treat  $f$  as a *black-box* and observe its behavior: We want to *synthesize* a simpler function with the same I/O behavior:

$(1, 1, 1)$	$\longrightarrow$	$f(x, y, z)$	$\longrightarrow$	3
$(2, 3, 1)$	$\longrightarrow$	$f(x, y, z)$	$\longrightarrow$	6
$(0, -7, 2)$	$\longrightarrow$	$f(x, y, z)$	$\longrightarrow$	-5

...

$$h(x, y, z) = x + y + z$$

# What is program synthesis?

The process of automatically constructing *programs* (code, expressions, etc.) that satisfy a given specification.

## Specification

Describe the expected behavior of the resulting synthesized candidate.

The implementation details are carried out by the synthesizer.

- Formal specification in some logic (e.g. first-order logic):

$$\forall x \in \mathbb{Z}/2^{64}\mathbb{Z}, \quad P(x) = x + 7$$

- A set of I/O pairs that describe the program behavior:

$$(0, 7), (-4, 3), (123, 130), (-368, -361), \dots$$

- A reference implementation (oracle) to generate I/O pairs.

# Synthesis approach

## Enumerative program synthesis (oracle-guided)

1. (Pre)generate an (offline) exhaustive list of potential candidates.
2. Generate a set of I/O pairs from the obfuscated code (oracle).
3. Select the candidates that match the oracle's I/O behavior.
4. If possible, verify semantic equivalence.

No candidates?

- Extend the pool of candidates (warning: exponential growth).

Multiple candidates?

- Check for semantic equivalence between them.
- Generate more I/O pairs.

# QSynth

Combines symbolic execution and enumerative program synthesis iteratively.

1. Split an obfuscated expression into smaller subexpressions.
2. Synthesize the subexpressions individually.
3. Reconstruct the overall simplified expression.

Paper: <https://profs.scienze.univr.it/~ceccato/papers/2020/bar2020.pdf>

# Tooling

Public implementations of the QSynth algorithm.

## **msynth**

Built on top of Miasm.

## **qsynthesis**

Built on top of Triton.



## Limitations

- Semantic complexity (e.g., non-toy cryptography).
- Non-determinism  $\neg(\text{!})$ .
- Point functions: constant output except for a single distinguished (small finite set of) input(s).

# Hands-on

# Thank you

## Archive

- [github.com/arnaugamez/talks](https://github.com/arnaugamez/talks)

## Reach out

- [arnaugamez.com](https://arnaugamez.com)
  - me <at> \$my\_domain
- [linkedin.com/in/arnaugamez](https://linkedin.com/in/arnaugamez)
- [infosec.exchange/@arnaugamez](https://infosec.exchange/@arnaugamez)