Hands-on binary (de)obfuscation

Arnau Gàmez i Montolio

FURA LABS

About

Hacker, Reverse Engineer & Mathematician

@arnaugamez

@FuraLabs

Today’s plan

Gentle introduction to modern binary (de)obfuscation.

Format: Brief lecture + Live coding

Goals: Understand the basic concepts and approaches, demystify the *magic* behind tools and techniques.

Topics: Mixed Boolean-Arithmetic, SMT analysis, symbolic execution, program synthesis.

Mixed Boolean-Arithmetic

Introduction

In a nutshell, a Mixed Boolean-Arithmetic (MBA) expression is composed of integer arithmetic operators, e.g. $(+, -, \times)$ and bitwise operators, e.g. $(\wedge, \vee, \oplus, \neg)$.

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

MBA expressions can be leveraged to **obfuscate the data-flow** of code by iteratively applying rewriting rules and function identities that complicate (obfuscate) the initial expression while **preserving its semantic behavior**.

Obfuscation idea

Combination of operators from these different fields **do not interact well together**: we have no rules (distributivity, factorization...) or general theory to deal with this mixing of operators.

Given an MBA expression E_1, we are interested in generating a **semantically equivalent** expression E_2 which is **syntactically more complex** than the initial expression E_1.

MBA rewriting

A chosen operator is rewritten with an equivalent MBA expression.

Example

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

SMT analysis

What is an SMT solver

TL;DR - SMT solvers can find *satisfying* solutions to a set of constraints.

SMT = SAT + Theories

We have three possible outcomes:

- *SAT*: A concrete assignment exists satisfying the constraints.
- *UNSAT*: There is no solution for the given set of constraints.
- *UNK*: Answer not found within the resource boundaries: timeout.

When a *SAT* outcome, a concrete assignment is known as a *model*.

We will work with the Z3 SMT solver, using its python API.

Basic usage

```c
bool checkKey(int64_t key)
{
  if (key > 10)
  {
    if (key - 23 < 10)
    {
      return (key * 3 < 100);
    }
  }
  return 0;
}
```

```python
from z3 import *

x = BitVec("x", 64)

solver = Solver()

solver.add(x > 10)
solver.add(x - 23 < 10)
solver.add(x * 3 < 100)

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

Program analysis with SMT solvers

SMT applications:

- Solve complex constraints
- Model counting
- **Check semantic equivalence**
- Input crafting
- Encode control-flow

Symbolic execution

What is a symbolic execution engine

Symbolic execution is *just* a **computer algebra system** targeting:

- programming languages
- assembly languages
- intermediate languages/representations (IL/IR)

The main idea is to transform the control-flow and data-flow of a program into *symbolic expressions*

Symbolic execution analysis

Extract path constraints to encode the branching conditions of a basic block with respect to the variables involved in it.

Extract formula for the value a variable will hold at some point in the program, with respect to the inputs defined at a starting point of the analysis (basic block, function, etc.).

Plugging an SMT solver

The symbolic execution engine is used to extract the formulae (constraints) for a given path branching condition. The constraints are fed into an SMT solver that can prove the feasibility of such paths (detect opaque predicates).

The symbolic execution engine is used to extract the formula for the return value of a function with respect to its input parameters. This formula can be fed to the SMT solver for a number of reasons:

- craft a valid input
- simplify the expression
- verify some property

Limitations

An attacker can increase arbitrarily the *syntactic* (algebraic) complexity of the obfuscated code, through MBA transformations.

→ SMT analysis does not scale well

Program synthesis

What is program synthesis

Program synthesis is the process of automatically constructing programs that satisfy a given specification.

By specification, we mean:

Somehow “telling the computer what to do”.

Let the implementation details to be carried by the *synthesizer*.

Specifying program behaviour

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

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

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

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

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

Problem statement

We want to recover (learn) the semantics of obfuscated code (expression) whose syntactic complexity has been arbitrarily increased to the point where an SMT solver is not *enough* to adequately simplify the obfuscated code (expression) into a *simple enough* representation of its semantics.

Synthesis approach

The nature of our problem leads to an **inductive oracle-guided program synthesis** style, using the obfuscated code as an I/O oracle:

- Generate a set of I/O pairs from the obfuscated code (oracle).

- Determine the best candidate program that matches the I/O behavior.

Limitations

Main limitations of (oracle-based) program synthesis:

- **Semantic complexity:** strong cryptography; confusion and diffusion
- **Non-determinism:** ¯_(ツ)_/¯
- **Point functions:** always return the same output for all inputs except for a single distinguished (*small* finite set of) input(s)

Live coding

More

Full version of this topic at *RingZer0 #BACK2VEGAS*:

Training: An Analytical Approach to Modern Binary Deobfuscation
Location: In-person @ Las Vegas, USA
Dates: August 06-09, 2022
Register: furalabs.com/r0-training