A review of modern code deobfuscation techniques

Arnau Gàmez i Montolio

Security Researcher

HITB LOCKDOWN 002 livestream

About

Arnau Gàmez i Montolio | 23

Graduated	|	Mathematics & Computer Engineering
President	|	@HackingLliure
Organizer	|	#r2con

Warning

This presentation may contain traces of assembly and maths

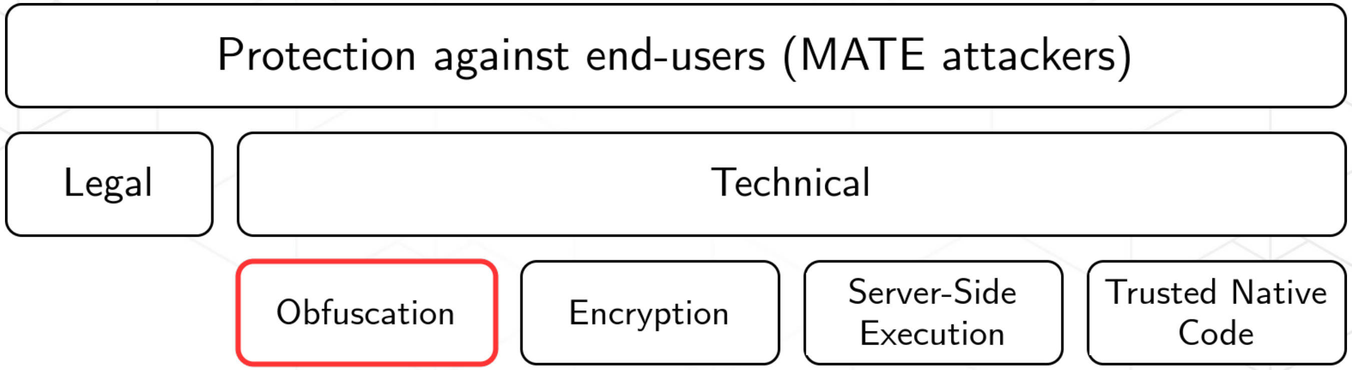
Contents

1) **Code obfuscation**
2) Mixed Boolean-Arithmetic
3) Program synthesis
4) r2syntia
5) Conclusions

Code obfuscation

Context

Technical protection against Man-At-The-End (MATE) attacks, where the attacker/analyst **has an instance** of the program and completely **controls the environment** where it is executed.



Protection against end-users (MATE attackers)

Legal

Technical

Obfuscation

Encryption

Server-Side Execution

Trusted Native Code

Code obfuscation

What

Transformation from a program *P* into a **functionally equivalent** program *P'* which is **harder to analyze and to extract information** than from *P*.

P → Obfuscation → *P'*

Code obfuscation

Who

Software protection

Malware threats

Code obfuscation

Why

~~Prevent~~ ***Complicate*** reverse engineering

- Intellectual property: algorithms/protocols in commercial software
- Digital Rights Management: access to software or digital content

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

Code obfuscation

How

Apply a transformation to **mess** (complicate) the program’s control-flow and/or data-flow at different **abstraction levels** (source code, compiled binary or an intermediate representation) and affecting different **target units** (whole program, function, basic block, instruction…).

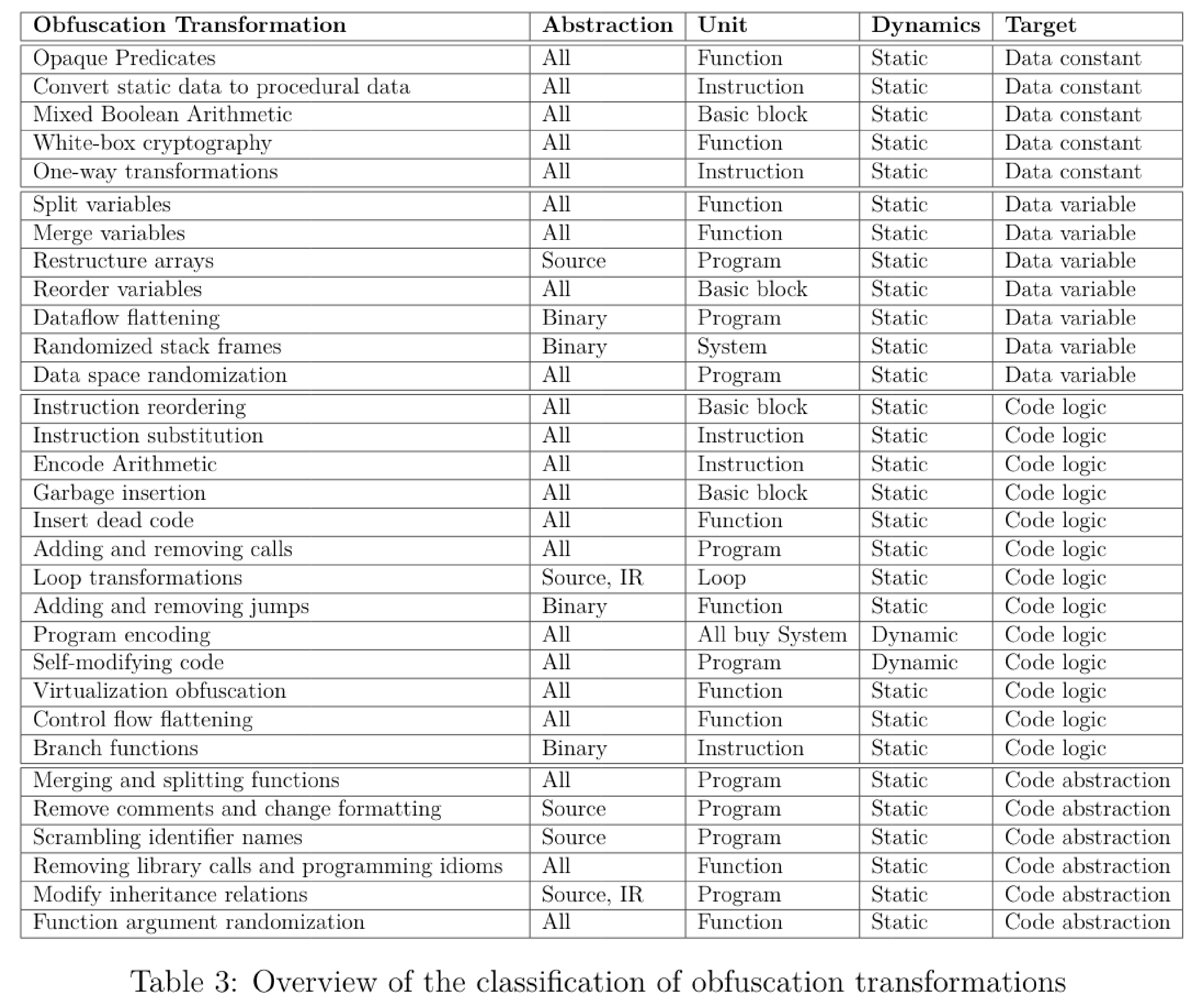
Many *weak* techniques can be combined to create a *hard* obfuscation transformation.

Co

How

Apply … rol-
flow a … de,
comp
differ
instru

Many … ation
trans



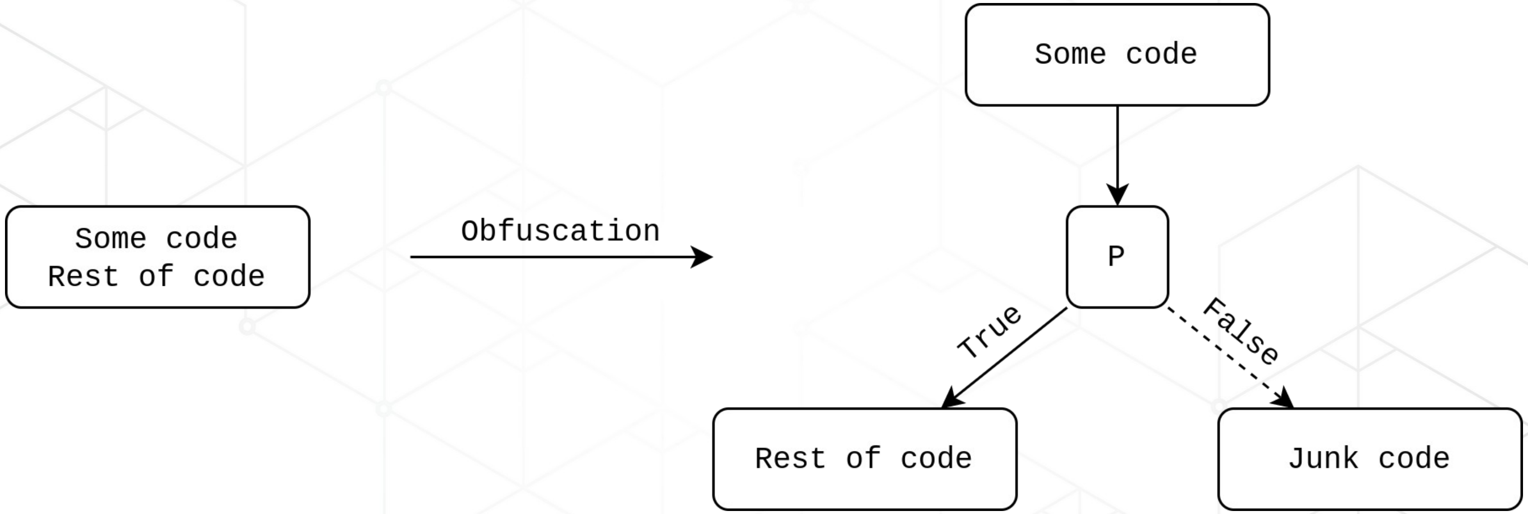
Obfuscation Transformation	Abstraction	Unit	Dynamics	Target
Opaque Predicates	All	Function	Static	Data constant
Convert static data to procedural data	All	Instruction	Static	Data constant
Mixed Boolean Arithmetic	All	Basic block	Static	Data constant
White-box cryptography	All	Function	Static	Data constant
One-way transformations	All	Instruction	Static	Data constant
Split variables	All	Function	Static	Data variable
Merge variables	All	Function	Static	Data variable
Restructure arrays	Source	Program	Static	Data variable
Reorder variables	All	Basic block	Static	Data variable
Dataflow flattening	Binary	Program	Static	Data variable
Randomized stack frames	Binary	System	Static	Data variable
Data space randomization	All	Program	Static	Data variable
Instruction reordering	All	Basic block	Static	Code logic
Instruction substitution	All	Instruction	Static	Code logic
Encode Arithmetic	All	Instruction	Static	Code logic
Garbage insertion	All	Basic block	Static	Code logic
Insert dead code	All	Function	Static	Code logic
Adding and removing calls	All	Program	Static	Code logic
Loop transformations	Source, IR	Loop	Static	Code logic
Adding and removing jumps	Binary	Function	Static	Code logic
Program encoding	All	All buy System	Dynamic	Code logic
Self-modifying code	All	Program	Dynamic	Code logic
Virtualization obfuscation	All	Function	Static	Code logic
Control flow flattening	All	Function	Static	Code logic
Branch functions	Binary	Instruction	Static	Code logic
Merging and splitting functions	All	Program	Static	Code abstraction
Remove comments and change formatting	Source	Program	Static	Code abstraction
Scrambling identifier names	Source	Program	Static	Code abstraction
Removing library calls and programming idioms	All	Function	Static	Code abstraction
Modify inheritance relations	Source, IR	Program	Static	Code abstraction
Function argument randomization	All	Function	Static	Code abstraction

Table 3: Overview of the classification of obfuscation transformations

Control-flow obfuscation

Opaque predicates

An opaque predicate is a specially crafted boolean expression *P* that always evaluates to either true or false.



Control-flow obfuscation

Control flow flattening

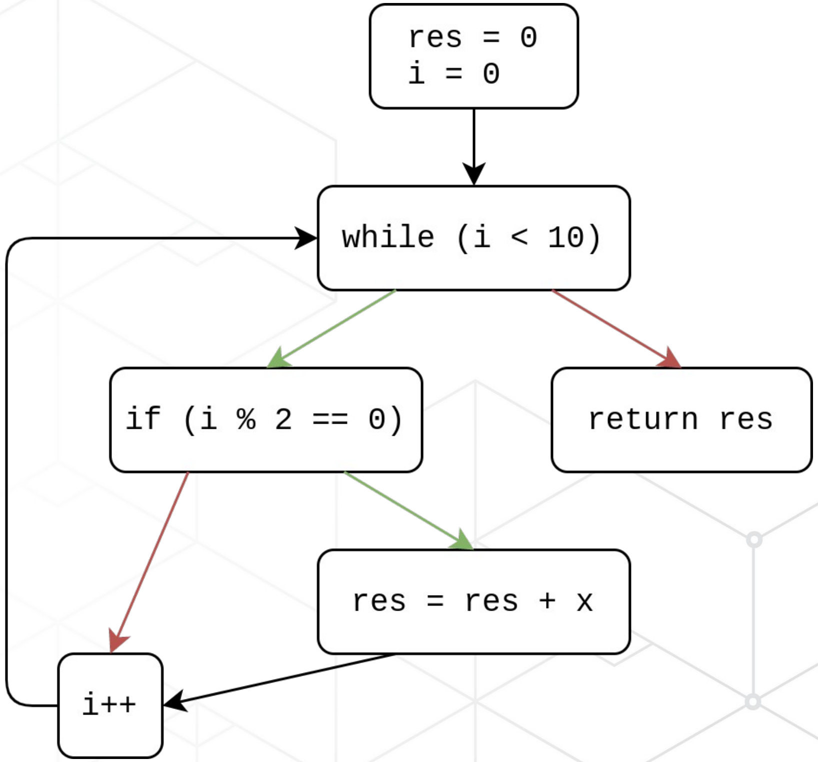
Change the structure of a function’s Control Flow Graph by replacing all control structures with a central and unique dispatcher.

Control-flow obfuscation

Control flow flattening

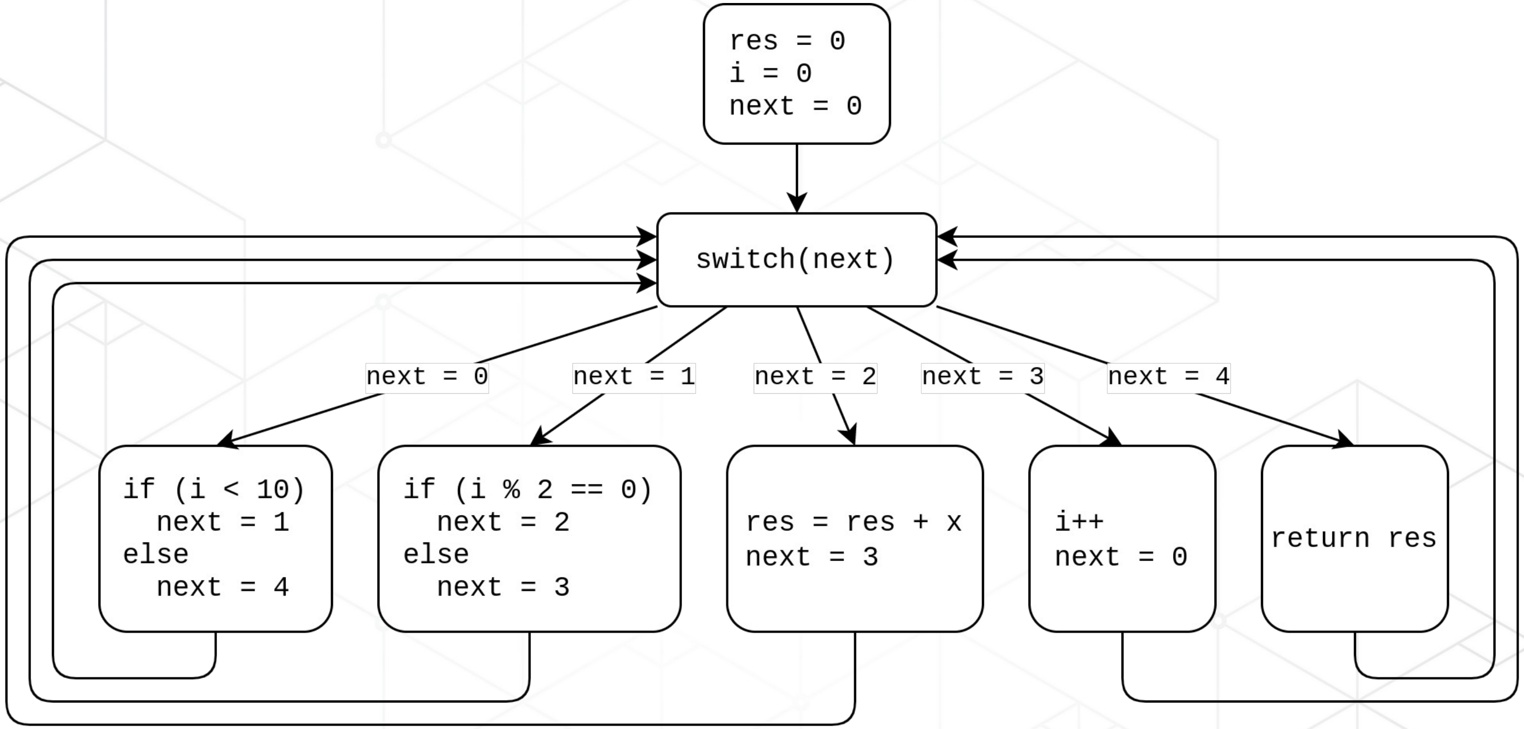
```
int f(int x) {
    int res = 0;
    int i = 0;

    while (i < 10) {
        if (i % 2 == 0)
            res = res + x;
        i++;
    }
    return res;
}
```



Control-flow obfuscation

Control flow flattening



Data-flow obfuscation

Dead code insertion

Deliberately insert instructions that will not have any effect in the computations’ outcome.

```
int f() {
    int x, y ,z;
    x = 1;
    y = 2;
    z = 3;
    x = y + 4;
    return x;
}
```

```
mov eax, 1
mob ebx, 2
mov ecx, 3
add ebx, 4
mov eax, ebx
```

Data-flow obfuscation

Encodings

Prevent a specific value to appear in clear at any point of the program execution. They are composed of an encoding function *f(x)* and its corresponding decoding function *g(x).*

f(x) = x – 0x1234
g(x) = x + 0x1234

```
...
sub eax, 0x1234              ; Apply encoding function
push eax                     ; Push eax on the stack
...
add dword [esp], 0x1234      ; Apply decoding function
...
pop ebx                      ; Retrieve decoded value
```

Data-flow obfuscation

Pattern substitution

Transform one or more adjacent instructions into a more complicated new sequence of instructions preserving semantic behavior.

```
push eax
```

```
lea esp, [esp - 4]
mov dword [esp], eax
```

```
push ebx
mov ebx, esp
xchg [esp], ebx
pop esp
mov dword [esp], eax
```

Data-flow obfuscation

Pattern substitution

Transform one or more adjacent instructions into a more complicated new sequence of instructions preserving semantic behavior.

```
push eax
```

```
lea esp, [esp - 4]
mov dword [esp], eax
```

```
push ebx
mov ebx, esp
xchg [esp], ebx
pop esp
mov dword [esp], eax
```

Code deobfuscation

What

Transformation from an obfuscated (piece of) program *P’* into a (piece of) program *P’’* which is **easier to analyze and to extract information** than from *P’*.

P’’ ← Deobfuscation ← *P’*

Code deobfuscation

Considerations

Ideally *P''* ≈ *P*, but this is rarely the case:

- **Lack of access** to original program *P* to compare.
- Interest in **specific parts** rather than whole program.
- Interest in **understanding** rather than reconstructing.

Contents

1) Code obfuscation

2) **Mixed Boolean-Arithmetic**

3) Program synthesis

4) r2syntia

5) Conclusions

MBA expressions

What

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

More formally...

MBA expressions

What

An expression E of the form $E = \sum_{i \in I} a_i \left(\prod_{j \in J_i} e_{i,j}(x_1, \ldots, x_t) \right)$ where the arithmetic sum and product are modulo 2^n, a_i are constants in $\mathbb{Z}/2^n\mathbb{Z}$, $e_{i,j}$ are bitwise expressions of variables $x_1, \ldots, x_t$ in $\{0,1\}^n$, $I \subset \mathbb{Z}$ and for all $i \in I$, $J_i \subset \mathbb{Z}$ are finite index sets, is a **polynomial Mixed Boolean-Arithmetic (MBA) expression**

A polynomial MBA expression of the form $E = \sum_{i \in I} a_i e_i(x_1, \ldots, x_t)$ is called a **linear MBA expression**.

MBA expressions

What

- Polynomial MBA:

$$E = 8458(x \vee y \wedge z)^3 ((xy) \wedge x \vee t) + x + 9(x \vee y)yz^3$$

- Linear MBA:

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

Obfuscation with MBA

MBA rewriting

A chosen operator is rewritten with an equivalent MBA expression.
The outcome of this process generates **rewriting rules**.

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

Obfuscation with MBA

Insertion of identities

Let e be any subexpression of the target expression being obfuscated. Then, we can write e as $f^{-1}(f(e))$ with f being any invertible function *(mod 2^n)*.

Obfuscation with MBA

Example

Consider $E_1 = x + y$ and the following functions f and f^{-1} on 8 bits:

$$f(x) = 39x + 23$$

$$f^{-1}(x) = 151x + 111$$

Consider e_1 obtained by applying the previous **rewriting rule** to E_1:

$$e_1 = (x \oplus y) + 2 \times (x \land y)$$

Obfuscation with MBA

Example

Then apply the **insertion of identities** produced by f and f^{-1}:

$$e_2 = f(e_1) = 39 \times e_1 + 23$$

$$E_2 = f^{-1}(e_2) = 151 \times e_2 + 111$$

Finally, expand E_2 to retrieve the obfuscated expression derived from the original expression $E_1 = x + y$:

$$E_2 = 151 \times (39 \times ((x \oplus y) + 2 \times (x \wedge y)) + 23) + 111$$

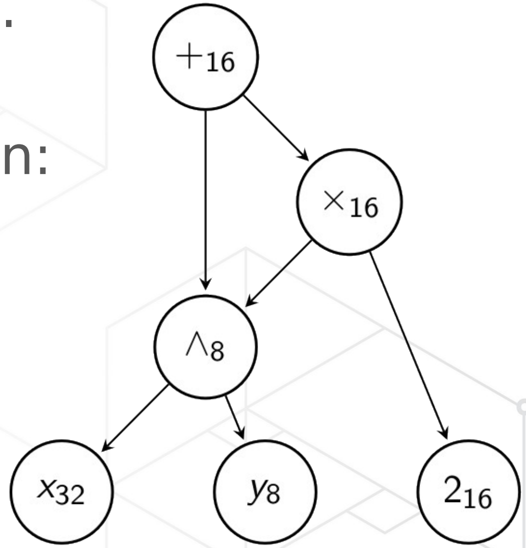
MBA expressions

Complexity metrics

We can represent an MBA expression as a Directed Acyclic Graph (DAG), which identifies common subexpressions.

Complexity metrics based on DAG representation:

- Number of nodes.
- MBA Alternation.
- Average bit-vector size.



DAG representation of $2 \times (x \wedge y) + (x \wedge y)$

Simplification

Bit-blasting approach

Find a canonical representation of MBA expressions:

- Represent MBA expressions as boolean expressions by computing the effect of each operation on each bit of the resulting value.
- Use Algebraic Normal Form (ANF) to guarantee unicity: expressions obtained will only contain *XOR* (⊕) and *AND* (∧) operators.

Simplification

Bit-blasting approach

Advantages:

- Transform the problem of MBA simplification into boolean expression simplification.

Drawbacks:

- Canonicalization can be very expensive (in memory and time).
- Identification of word-level expressions from boolean expressions is far from trivial.
- Scalability issues for large number of bits.

Simplification

Symbolic approach

Find an equivalent, but simpler form:

- Use existing simplification techniques for parts of the MBA expression containing only one type of operator.
- Use a term rewriting approach to create the missing link between subexpressions alternating different types of operators.
- Rewriting rules for deobfuscation can be obtained by inverting the direction of rewriting rules used for obfuscation.

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

Simplification

Symbolic approach

Advantages:

- The simplification is not impeded by an increasing number of bits.
- The representation of the expressions is far smaller than the representation in the bit-blasting approach.

Drawbacks:

- Very sensible to the size of the obfuscated expression.
- Highly dependent on the chosen set of rewriting rules.

Contents

Program synthesis

Motivating example

Consider the following function (an obfuscated MBA expression):

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

We can treat it as a **black-box** and observe its behavior:

$$(1,1,1) \longrightarrow \boxed{f(x,y,z)} \longrightarrow 3$$

$$(2,3,1) \longrightarrow \boxed{f(x,y,z)} \longrightarrow 6$$

$$(0,-7,2) \longrightarrow \boxed{f(x,y,z)} \longrightarrow -5$$

$$\dots$$

Program synthesis

Motivating example

$$
\begin{array}{rcccl}
(1,1,1) & \longrightarrow & \boxed{f(x,y,z)} & \longrightarrow & 3 \\
(2,3,1) & \longrightarrow & \boxed{f(x,y,z)} & \longrightarrow & 6 \\
(0,-7,2) & \longrightarrow & \boxed{f(x,y,z)} & \longrightarrow & -5 \\
 & & \dots & &
\end{array}
$$

Our objective is to *learn* (*synthesize*) a simpler function with the same I/O behavior:

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

Program synthesis

What

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.*

Program synthesis

Specification

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

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

- 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.

Program 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 observed I/O behavior.

Program synthesis

Practical considerations

- To construct candidate programs, we define a **context-free grammar** that encompasses the primitive components (terminals) and the ways to combine them (production rules).
- **Set boundaries** that delimit the program synthesis task (I/O pairs, terminals and derivations of the context-free grammar) and ensure that it terminates (iterations and time).
- Decide when a synthesized candidate is *valid enough* and whether to introduce some kind of equivalence checking.

Existing work

Syntia (2017)

Monte Carlo Tree Search (MCTS) based stochastic program synthesis.

- Convert the problem of finding a candidate program into a stochastic optimization problem.
- At each iteration we generate intermediate results instead of actual candidate programs.
- Evolve towards a global optima (best candidate program) guided by a cost function.

Existing work

Syntia (2017)

A public (and open source) implementation is available :)

Existing work

QSynth (2020)

Offline enumerative program synthesis.

- Given a context-free grammar, generate *all* programs up to a certain number of derivations.
- Create *offline* lookup tables mapping each candidate program to its I/O behavior.
- Perform an exhaustive search for candidate programs matching the oracle’s I/O behavior.

Existing work

QSynth (2020)

Most significant contribution (IMHO): **Split** an obfuscated expression into smaller subexpressions, **synthesize** them individually and then **reconstruct** the total simplified expression.

Existing work

QSynth (2020)

Unfortunately, there is no public implementation available :(

Program synthesis

Limitations

- Semantic complexity.
- Non-determinism.
- Point functions.

Contents

1) Code obfuscation
2) Mixed Boolean-Arithmetic
3) Program synthesis
4) **r2syntia**
5) Conclusions

r2syntia

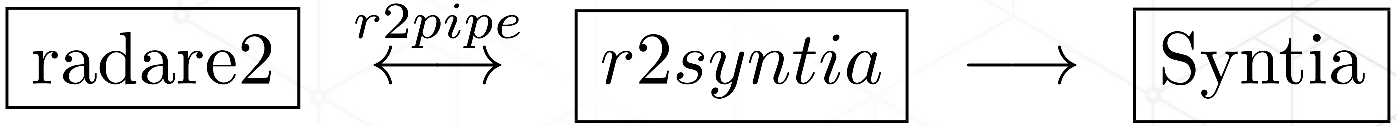
Components

- radare2: fully-fledged reverse engineering framework.
- ESIL: radare2 emulation engine.
- Syntia: program synthesis based framework for deobfuscation.

r2syntia

Integration

- Call r2syntia from an active radare2 shell were we are performing the analysis of a binary that contains obfuscated code.
- r2syntia leverages ESIL to generate the I/O pairs of values for the specified variables (registers and memory locations).
- Invoke Syntia from r2syntia with the generated I/O pairs.



radare2 $\xleftrightarrow{r2pipe}$ *r2syntia* ⟶ Syntia

r2syntia

Result

Synthesize the code semantics of the output variable (register or memory location) with respect to the input variables (registers or memory locations).

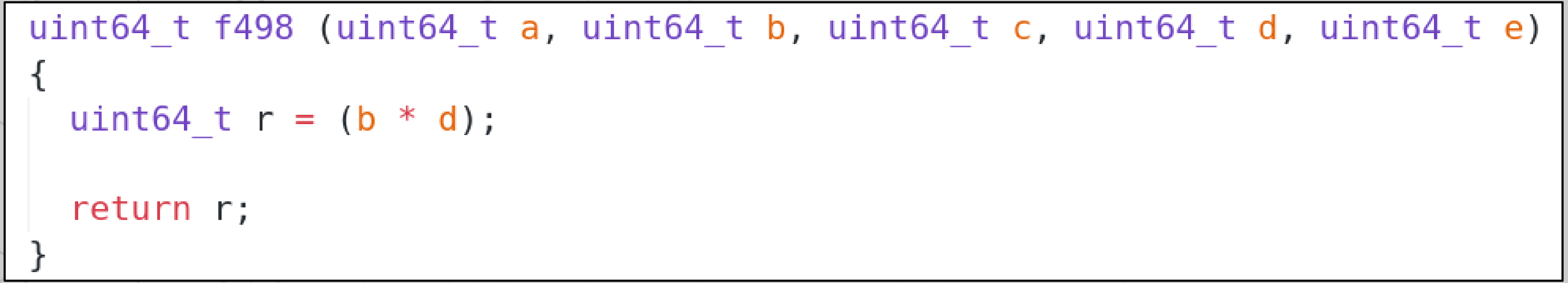
r2syntia

Guided example

$\boxed{\text{Non-obfuscated } C}$

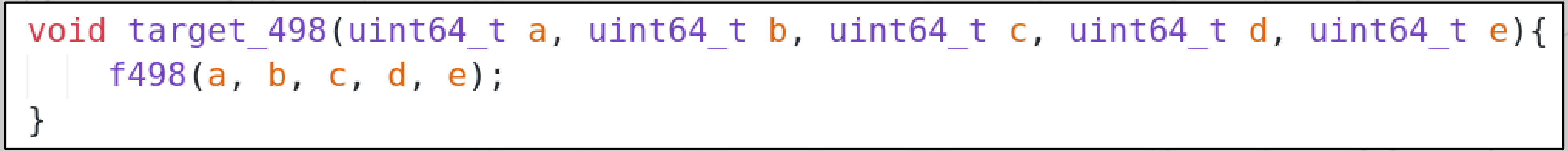
r2syntia

Guided example



```
uint64_t f498 (uint64_t a, uint64_t b, uint64_t c, uint64_t d, uint64_t e)
{
  uint64_t r = (b * d);

  return r;
}
```



```
void target_498(uint64_t a, uint64_t b, uint64_t c, uint64_t d, uint64_t e){
    f498(a, b, c, d, e);
}
```

r2syntia

Guided example

Obfuscate the functions with Tigress v2.2:

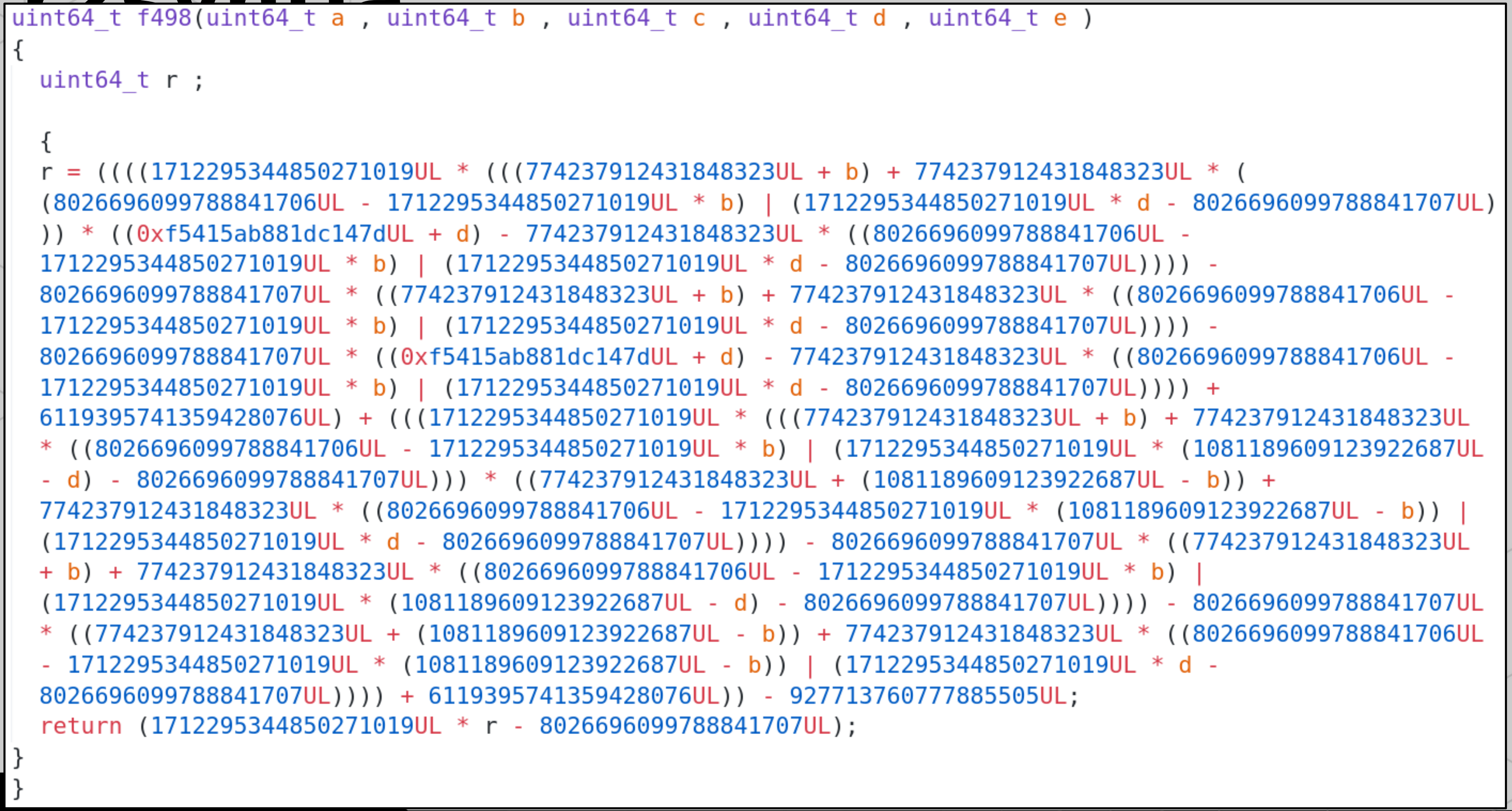
- *EncodeArithmetic*: replaces the original expression with an equivalent (more complicated) MBA expression.
- *EncodeData*: encodes integer arguments before calling the function and decodes them at return.

r2syntia

Guided example

$$\boxed{\text{Non-obfuscated } C} \xrightarrow{Tigress} \boxed{\text{Obfuscated } C}$$

r2syntia

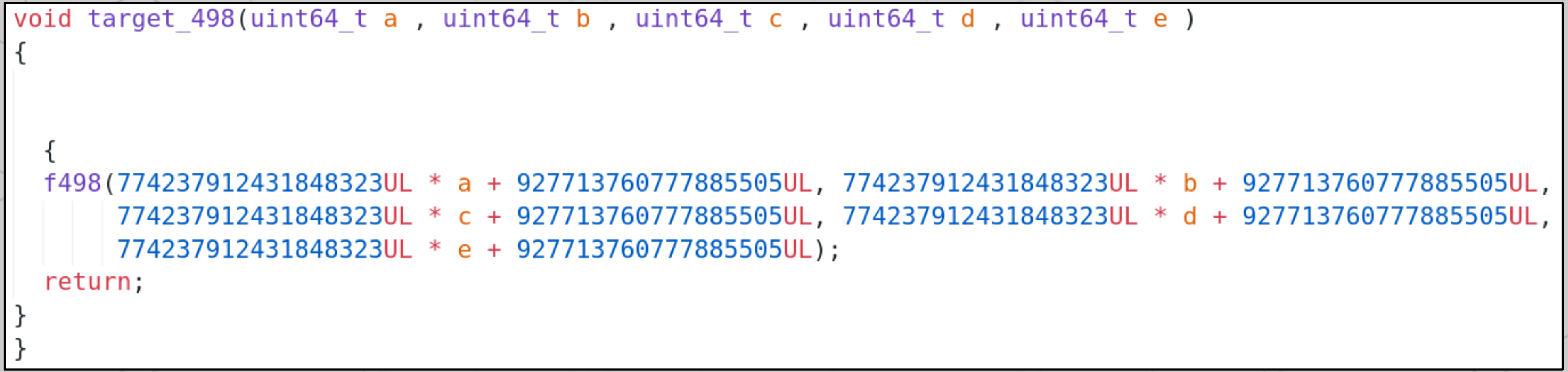


```
uint64_t f498(uint64_t a , uint64_t b , uint64_t c , uint64_t d , uint64_t e )
{
  uint64_t r ;

  {
  r = ((((1712295344850271019UL * (((774237912431848323UL + b) + 774237912431848323UL * (
  (8026696099788841706UL - 1712295344850271019UL * b) | (1712295344850271019UL * d - 8026696099788841707UL)
  )) * ((0xf5415ab881dc147dUL + d) - 774237912431848323UL * ((8026696099788841706UL -
  1712295344850271019UL * b) | (1712295344850271019UL * d - 8026696099788841707UL)))) -
  8026696099788841707UL * ((774237912431848323UL + b) + 774237912431848323UL * ((8026696099788841706UL -
  1712295344850271019UL * b) | (1712295344850271019UL * d - 8026696099788841707UL)))) -
  8026696099788841707UL * ((0xf5415ab881dc147dUL + d) - 774237912431848323UL * ((8026696099788841706UL -
  1712295344850271019UL * b) | (1712295344850271019UL * d - 8026696099788841707UL)))) +
  6119395741359428076UL) + (((1712295344850271019UL * (((774237912431848323UL + b) + 774237912431848323UL
  * ((8026696099788841706UL - 1712295344850271019UL * b) | (1712295344850271019UL * (1081189609123922687UL
  - d) - 8026696099788841707UL))) * ((774237912431848323UL + (1081189609123922687UL - b)) +
  774237912431848323UL * ((8026696099788841706UL - 1712295344850271019UL * (1081189609123922687UL - b)) |
  (1712295344850271019UL * d - 8026696099788841707UL)))) - 8026696099788841707UL * ((774237912431848323UL
  + b) + 774237912431848323UL * ((8026696099788841706UL - 1712295344850271019UL * b) |
  (1712295344850271019UL * (1081189609123922687UL - d) - 8026696099788841707UL)))) - 8026696099788841707UL
  * ((774237912431848323UL + (1081189609123922687UL - b)) + 774237912431848323UL * ((8026696099788841706UL
  - 1712295344850271019UL * (1081189609123922687UL - b)) | (1712295344850271019UL * d -
  8026696099788841707UL)))) + 6119395741359428076UL)) - 927713760777885505UL;
  return (1712295344850271019UL * r - 8026696099788841707UL);
}
}
```

r2syntia

Guided example



```
void target_498(uint64_t a , uint64_t b , uint64_t c , uint64_t d , uint64_t e )
{


  {
  f498(774237912431848323UL * a + 927713760777885505UL, 774237912431848323UL * b + 927713760777885505UL,
       774237912431848323UL * c + 927713760777885505UL, 774237912431848323UL * d + 927713760777885505UL,
       774237912431848323UL * e + 927713760777885505UL);
  return;
}
}
```

r2syntia

Guided example

$$\boxed{\text{Non-obfuscated } C} \xrightarrow{Tigress} \boxed{\text{Obfuscated } C} \xrightarrow{Compiler} \boxed{\text{Obfuscated } ASM}$$

Demo: Obfuscated ASM

r2syntia

Guided example

If we address the task of deobfuscating this code through a symbolic execution approach, we could obtain an expression representing the return value of the function we are analyzing, with respect to the input variables.

Let’s observe the resulting obfuscated MBA expression obtained using Metasm.

r

G

If

ex

re

in

Le

us

File: **obf_mba_expr_symbolic_498**

```
((17c34b8b445afb2bh*((((((17c34b8b445afb2bh*(((0abea5477e23eb83h*((-((17c34b8b445afb2bh*((0abea5477e23eb83h
*rsi)&0ffffffffffffffffh)+131f0149f036dde6f648d9353ed62ebh)&0ffffffffffffffffh)+6f648d9353ed62eah)|(((0e83c
b474bba504d5h*((0abea5477e23eb83h*rcx)&0ffffffffffffffffh)+0badf6ab6d64e962909b726cac129d15h)&0ffffffffffff
ffffh)+6f648d9353ed62eah)))&0ffffffffffffffffh)+((0abea5477e23eb83h*rsi)&0ffffffffffffffffh)+0cdfe6c00c6857
41h))&0ffffffffffffffffh)+1)*(((0abea5477e23eb83h*((-((0e83cb474bba504d5h*((0abea5477e23eb83h*rsi)&0fffffff
fffffffffh)+0badf6ab6d64e962909b726cac129d15h)&0ffffffffffffffffh)+909b726cac129d15h)|(((17c34b8b445afb2bh*
((0abea5477e23eb83h*rcx)&0ffffffffffffffffh)+131f0149f036dde6f648d9353ed62ebh)&0ffffffffffffffffh)+909b726c
ac129d15h)))&0ffffffffffffffffh)-((0abea5477e23eb83h*rsi)&0ffffffffffffffffh)+0cdfe6c00c685741h))&0ffffffff
ffffffffh)-((6f648d9353ed62ebh*(((0abea5477e23eb83h*((-((17c34b8b445afb2bh*((0abea5477e23eb83h*rsi)&0ffffff
ffffffffffh)+131f0149f036dde6f648d9353ed62ebh)&0ffffffffffffffffh)+6f648d9353ed62eah)|(((0e83cb474bba504d5h
*((0abea5477e23eb83h*rcx)&0ffffffffffffffffh)+0badf6ab6d64e962909b726cac129d15h)&0ffffffffffffffffh)+6f648d
9353ed62eah)))&0ffffffffffffffffh)+((0abea5477e23eb83h*rsi)&0ffffffffffffffffh)+0cdfe6c00c685741h))&0ffffff
ffffffffffh)-((6f648d9353ed62ebh*(((0abea5477e23eb83h*((-((0e83cb474bba504d5h*((0abea5477e23eb83h*rsi)&0fff
fffffffffffffh)+0badf6ab6d64e962909b726cac129d15h)&0ffffffffffffffffh)+909b726cac129d15h)|(((17c34b8b445afb
2bh*((0abea5477e23eb83h*rcx)&0ffffffffffffffffh)+131f0149f036dde6f648d9353ed62ebh)&0ffffffffffffffffh)+909b
726cac129d15h)))&0ffffffffffffffffh)-((0abea5477e23eb83h*rsi)&0ffffffffffffffffh)-0cdfe6c00c685741h))&0ffff
ffffffffffffh)+(((((17c34b8b445afb2bh*(((0abea5477e23eb83h*((-((17c34b8b445afb2bh*((0abea5477e23eb83h*rsi)&
0ffffffffffffffffh)+131f0149f036dde6f648d9353ed62ebh)&0ffffffffffffffffh)+6f648d9353ed62eah)|(((17c34b8b445
afb2bh*((0abea5477e23eb83h*rcx)&0ffffffffffffffffh)+131f0149f036dde6f648d9353ed62ebh)&0ffffffffffffffffh)+9
09b726cac129d15h)))&0ffffffffffffffffh)+((0abea5477e23eb83h*rsi)&0ffffffffffffffffh)+0cdfe6c00c685741h))&0f
fffffffffffffffh)+1)*(((0abea5477e23eb83h*rcx)&0ffffffffffffffffh)-((0abea5477e23eb83h*((-((17c34b8b445afb2
bh*((0abea5477e23eb83h*rsi)&0ffffffffffffffffh)+131f0149f036dde6f648d9353ed62ebh)&0ffffffffffffffffh)+6f648
d9353ed62eah)|(((17c34b8b445afb2bh*((0abea5477e23eb83h*rcx)&0ffffffffffffffffh)+131f0149f036dde6f648d9353ed
62ebh)&0ffffffffffffffffh)+909b726cac129d15h)))&0ffffffffffffffffh)+1022141788e446bbeh))&0ffffffffffffffffh
)-((6f648d9353ed62ebh*(((0abea5477e23eb83h*((-((17c34b8b445afb2bh*((0abea5477e23eb83h*rsi)&0fffffffffffffff
fh)+131f0149f036dde6f648d9353ed62ebh)&0ffffffffffffffffh)+6f648d9353ed62eah)|(((17c34b8b445afb2bh*((0abea54
77e23eb83h*rcx)&0ffffffffffffffffh)+131f0149f036dde6f648d9353ed62ebh)&0ffffffffffffffffh)+909b726cac129d15h
)))&0ffffffffffffffffh)+((0abea5477e23eb83h*rsi)&0ffffffffffffffffh)+0cdfe6c00c685741h))&0ffffffffffffffffh
)-((6f648d9353ed62ebh*(((0abea5477e23eb83h*rcx)&0ffffffffffffffffh)-((0abea5477e23eb83h*((-((17c34b8b445afb
2bh*((0abea5477e23eb83h*rsi)&0ffffffffffffffffh)+131f0149f036dde6f648d9353ed62ebh)&0ffffffffffffffffh)+6f64
8d9353ed62eah)|(((17c34b8b445afb2bh*((0abea5477e23eb83h*rcx)&0ffffffffffffffffh)+131f0149f036dde6f648d9353e
d62ebh)&0ffffffffffffffffh)+909b726cac129d15h)))&0ffffffffffffffffh)+0cdfe6c00c685741h))&0ffffffffffffffffh
)))&0ffffffffffffffffh)+909b726cac129d15h
```

r2syntia

Guided example

Observe that the syntactic complexity of this expression makes it unapproachable and useless to derive any understanding from it.

However, its semantic behavior is fairly simple. Thus, we can leverage r2syntia to extract the actual code semantics.

Demo: r2syntia

r2syntia

Publication

Already published!

Go play with it: https://github.com/arnaugamez/r2syntia

Contents

1) Code obfuscation

2) Mixed Boolean-Arithmetic

3) Program synthesis

4) r2syntia

5) **Conclusions**

Conclusions

Takeaways

- MBA expressions can be leveraged to obfuscate the data-flow of a program.
- Current deobfuscation techniques (e.g. symbolic execution) to address simplification of this type of data-flow obfuscation are limited by being strongly tied to syntactic complexity.
- Novel program synthesis approaches allow to **reason about the semantics** of the obfuscated code (instead of syntax).

Conclusions

Future work

Theoretical continuation:

- Further study and formalization of MBA expressions’ treatment.

Practical continuation:

- Improve r2syntia (WIP).
- Implement an (open source) solution for subexpressions’ synthesis.
- Detect patterns and memorize synthesized tasks.

Conclusions

References I

- Bruce Dang et al. *“Practical Reverse Engineering: X86, X64, ARM, Windows Kernel, Reversing Tools, and Obfuscation”* (chapter 5).
- Banescu et al. *“A Tutorial on Software Obfuscation”*.
- Christian Collberg and Jasvir Nagra. *“Surreptitious Software: Obfuscation, Water-marking, and Tamperproofing for Software Protection”.*
- *“Advanced Binary Deobfuscation”*: https://github.com/malrev/ABD

Conclusions

References II

- Ninon Eyrolles. *“Obfuscation with Mixed Boolean-Arithmetic Expressions: reconstruction, analysis and simplification tools”*.
- Tim Blazytko et al. *“Syntia: Synthesizing the Semantics of Obfuscated Code”*.
- Robin David, Luigi Coniglio, and Mariano Ceccato. *“QSynth - A Program Synthesis based approach for Binary Code Deobfuscation”*.

Thank You!

HITB LOCKDOWN 002 livestream

Arnau Gàmez i Montolio, *me ‹ατ› arnaugamez ‹δοτ› com*

@arnaugamez