Ninon Eyrolles
neyrolles@quarkslab.com
Louis Goubin
louis.goubin@uvsq.fr
Marion Videau
mvideau@quarkslab.com

Defeating MBA-based Obfuscation

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
université PARIS-SACLAY

quarkslab
SECURING EVERY BIT OF YOUR DATA

Prelude

```
def foo(x):
    v0 = (x*0xE5 + 0xF7)
    v3 = ((((v0*0x26)+0x55)&0xFE)+(v0*0xED)+0xD6)
    v4 = ((((((- (v3*0x2))+0xFF)&0xFE)+v3)*0x03)+0x4D)
    v5 = (((((v4*0x56)+0x24)&0x46)*0x4B)+(v4*0xE7)+0x76)
    v7 = ((((v5*0x3A)+0xAF)&0xF4)+(v5*0x63)+0x2E)
    v6 = (v7&0x94)
    v8 = ((((v6+v6+(- (v7&0xFF)))*0x67)+0xD))
    result = ((v8*0x2D)+(((v8*0xAE)|0x22)*0xE5)+0xC2)
    result = (0xed*(result-0xF7))&0xFF
    return result
```

Prelude

```
def foo(x):
    return ((x ^ 0x5C) & 0xFF)
```

Introduction

- Data-flow obfuscation with *mixed* expressions
- Defining a notion of *simplification* is an issue
- Assessing the resilience of the MBA obfuscation

Table of Contents

Background
- MBA Obfuscation
- Expression Simplification

Utility of MBA in Obfuscation

Our Contribution: MBA Simplification

Conclusion

Definition

Mixed Boolean-Arithmetic expression

An expression mixing arithmetic operators $(+, -, \times)$ and bitwise/boolean operators $(\wedge, \vee, \oplus, \neg)$.

Definition

Mixed Boolean-Arithmetic expression

An expression mixing arithmetic operators $(+, -, \times)$ and bitwise/boolean operators $(\wedge, \vee, \oplus, \neg)$.

Example

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

Definition

Mixed Boolean-Arithmetic expression

An expression mixing arithmetic operators $(+, -, \times)$ and bitwise/boolean operators $(\wedge, \vee, \oplus, \neg)$.

Example

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

- Already exists in cryptography (without the name)
- Defined in *Information Hiding in Software with Mixed Boolean-Arithmetic* [Zhou et al. 2007]

Definition

Mixed Boolean-Arithmetic expression

An expression mixing arithmetic operators $(+, -, \times)$ and bitwise/boolean operators $(\wedge, \vee, \oplus, \neg)$.

Example

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

- Already exists in cryptography (without the name)
- Defined in *Information Hiding in Software with Mixed Boolean-Arithmetic* [Zhou et al. 2007]

NB: $f(x, y) = x + y$

MBA Obfuscation

Expression Obfuscation

- **Rewritings**: $x + y \rightarrow (x \oplus y) + 2 \times (x \wedge y)$
- **Encodings**: $x \rightarrow 237 \times (229x + 247) + 85$ (on 8 bits)
- Those two steps are applied iteratively

```python
def foo(x):
    v0 = (x*0xE5 + 0xF7)
    v3 = ((((v0*0x26)+0x55)&0xFE)+(v0*0xED)+0xD6)
    v4 = ((((((- (v3*0x2))+0xFF)&0xFE)+v3)*0x03)+0x4D)
    v5 = (((((v4*0x56)+0x24)&0x46)*0x4B)+(v4*0xE7)+0x76)
    v7 = ((((v5*0x3A)+0xAF)&0xF4)+(v5*0x63)+0x2E)
    v6 = (v7&0x94)
    v8 = ((((v6+v6+(- (v7&0xFF)))*0x67)+0xD))
    result = ((v8*0x2D)+(((v8*0xAE)|0x22)*0xE5)+0xC2)
    result = (0xed*(result-0xF7))&0xFF
    return result
```

The Issue of Simplification

Two types of simplification

- canonical representation (unique for equivalent objects but does not always exist)
- simpler form (context-dependent)

But canonical does not always mean simplest form!

The Issue of Simplification

Two types of simplification

- canonical representation (unique for equivalent objects but does not always exist)
- simpler form (context-dependent)

But canonical does not always mean simplest form!

- Simpler can mean: cheaper to store, easier to read, quicker to compute...

The Issue of Simplification

Two types of simplification

- canonical representation (unique for equivalent objects but does not always exist)
- simpler form (context-dependent)

But canonical does not always mean simplest form!

- Simpler can mean: cheaper to store, easier to read, quicker to compute...
- Obfuscation is designed to counter human and automatic analysis

The Issue of Simplification

Two types of simplification

- canonical representation (unique for equivalent objects but does not always exist)
- simpler form (context-dependent)

But canonical does not always mean simplest form!

- Simpler can mean: cheaper to store, easier to read, quicker to compute...
- Obfuscation is designed to counter human and automatic analysis
- We need to assess the simplicity of MBA expressions specifically

The Issue of Simplification

Two types of simplification

- canonical representation (unique for equivalent objects but does not always exist)
- simpler form (context-dependent)

But canonical does not always mean simplest form!

- Simpler can mean: cheaper to store, easier to read, quicker to compute...
- Obfuscation is designed to counter human and automatic analysis
- We need to assess the simplicity of MBA expressions specifically
- The notion of simplicity can also depend on the simplification algorithm

Arithmetic Simplification

The expanded form is canonical, but not always the most readable.

$$(x-3)^2 - x^2 + 7x - 7 = x + 2$$
$$(1+x)^{100} = 1 + 100x + \cdots + 100x^{99} + x^{100}$$

Most computer algebra software provide different forms.

Boolean Simplification

Several canonical forms available: CNF, DNF, ANF...

Circuit simplification

Can reduce:

- the number of gates
- the depth of the circuit
- the fan-out of the gates
- ...

Example:

$$
\begin{aligned}
& (A \wedge B) \vee (B \wedge C \wedge (B \vee C)) \\
\text{CNF: } & (B \wedge (A \vee C)) \\
\text{DNF: } & ((A \wedge B) \vee (B \wedge C))
\end{aligned}
$$

Mixed Simplification

Attacks on the MBA obfuscation for **opaque constants** in *Effectiveness of Synthesis in Concolic Deobfuscation* [Biondi et al. 2015]:

- approach using SMT solvers
- algebraic simplification technique
- drill-and-join synthesis method

All these attacks are very dependent on the obfuscation technique, which is different for constant and operator obfuscation.

Mixed Simplification

Attacks on the MBA obfuscation for **opaque constants** in *Effectiveness of Synthesis in Concolic Deobfuscation* [Biondi et al. 2015]:

- approach using SMT solvers
- algebraic simplification technique
- drill-and-join synthesis method

All these attacks are very dependent on the obfuscation technique, which is different for constant and operator obfuscation.

We focus on the expression obfuscation.

Table of Contents

Background

Utility of MBA in Obfuscation
- First Issues
- Analysis difficulties

Our Contribution: MBA Simplification

Conclusion

First Issues

- Incompatibility of operators: no general rules for mixed expressions
- Compiler optimisations are inefficient
- Symbolic computation software rarely support bitwise operators
- SMT solvers often implement *bit-vector logic*, but goals differ

Analysis Difficulties

Reverse engineering context

- *Compilation:* the obfuscation very probably occured during compilation, optimisation passes may have been applied
- *Extraction:* a method is needed to obtain an expression from the assembly language (e.g. *symbolic execution*)

Table of Contents

Background

Utility of MBA in Obfuscation

Our Contribution: MBA Simplification
- Simplicity Metrics
- Algorithm
- SSPAM

Conclusion

Expression Representation

Term graph representation: acyclic graph

- leaves are constant numbers or variables, other nodes are operators
- an edge from o to e means e is an operand of operator o
- there is only one root node
- common expressions are *shared*

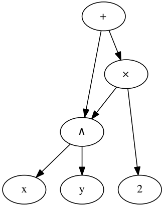


Figure : Term graph for the expression $2 \times (x \wedge y) + (x \wedge y)$.

Sharing of Subgraphs

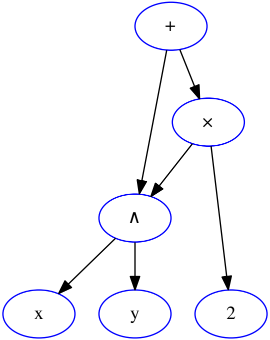
```
def foo(x):
    return ((237 * ((((((((((((((((((((((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254)
    + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 86) + 36) & 70) * 75) + (((((((- ((((((((x * 229)
    + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247)
    * 237)) + 214)) * 3) + 77) * 231)) + 118) * 58) + 175) & 244) + ((((((((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229)
    + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 86) + 36)
    & 70) * 75) + (((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229)
    + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 231)) + 118) * 99)) + 46) & 148) + ((((((((((((((((((- ((((((((x * 229)
    + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247)
    * 237)) + 214)) * 3) + 77) * 86) + 36) & 70) * 75) + (((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2))
    + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 231)) + 118) * 58) + 175) & 244)
    + ((((((((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38)
    + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 86) + 36) & 70) * 75) + (((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x
    * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 231))
    + 118) * 99)) + 46) & 148)) + (- (((((((((((((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255)
    & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 86) + 36) & 70) * 75) + (((((((- ((((((((x
    * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229)
    + 247) * 237)) + 214)) * 3) + 77) * 231)) + 118) * 58) + 175) & 244) + ((((((((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229)
    + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 86) + 36)
    & 70) * 75) + (((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247)
    * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 231)) + 118) * 99)) + 46) & 255))) * 103) + 13) * 45) + (((((((((((((((((((((((((
    - ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254)
    + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 86) + 36) & 70) * 75) + (((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247)
    * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 231)) + 118)
    * 58) + 175) & 244) + ((((((((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x
    * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 86) + 36) & 70) * 75) + (((((((- ((((((((x * 229) + 247) * 38)
    + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237))
    + 214)) * 3) + 77) * 231)) + 118) * 99)) + 46) & 148) + ((((((((((((((((( - ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237))
    + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 86) + 36) & 70) * 75)
    + (((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85)
    & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 231)) + 118) * 58) + 175) & 244) + ((((((((((((- ((((((((x * 229) + 247) * 38) + 85)
    & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3)
    + 77) * 86) + 36) & 70) * 75) + (((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254)
    + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 231)) + 118) * 99)) + 46) & 148))
    + (- (((((((((((((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229)
    + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 86) + 36) & 70) * 75) + (((((((- ((((((((x * 229) + 247) * 38) + 85)
    & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3)
    + 77) * 231)) + 118) * 58) + 175) & 244) + ((((((((((((- ((((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255)
    & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214)) * 3) + 77) * 86) + 36) & 70) * 75) + (((((((- ((((((((x
    * 229) + 247) * 38) + 85) & 254) + (((x * 229) + 247) * 237)) + 214) * 2)) + 255) & 254) + (((((((x * 229) + 247) * 38) + 85) & 254) + (((x * 229)
    + 247) * 237)) + 214)) * 3) + 77) * 231)) + 118) * 99)) + 46) & 255))) * 103) + 13) * 174) | 34) * 229)) + 194) - 247)) & 255)
```

Sharing of Subgraphs

```python
def foo(x):
    v0 = (x*0xE5 + 0xF7)
    v3 = ((((v0*0x26)+0x55)&0xFE)+(v0*0xED)+0xD6)
    v4 = ((((((- (v3*0x2))+0xFF)&0xFE)+v3)*0x03)+0x4D)
    v5 = (((((v4*0x56)+0x24)&0x46)*0x4B)+(v4*0xE7)+0x76)
    v7 = ((((v5*0x3A)+0xAF)&0xF4)+(v5*0x63)+0x2E)
    v6 = (v7&0x94)
    v8 = ((((v6+v6+(- (v7&0xFF)))*0x67)+0xD))
    result = ((v8*0x2D)+(((v8*0xAE)|0x22)*0xE5)+0xC2)
    result = (0xed*(result-0xF7))&0xFF
    return result
```

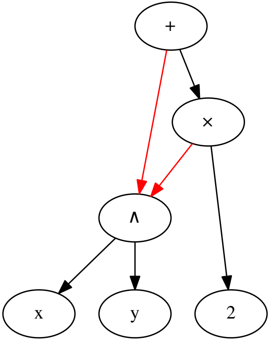
Simplicity Metrics (1)

Number of nodes: reducing the size improves the readability and makes it easier to manipulate for any software.



Simplicity Metrics (2)

MBA alternance: number of edges linking two operators of different types (arithmetic or boolean).



Algorithm

Step one: MBA rewriting

Invert the rewriting process of the obfuscation. For example:

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

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

Step two: arithmetic simplification

Use existing arithmetic simplification techniques. For example, on 8 bits:

$$237 \times (229x + 247) + 85 = x$$

Detailed example

$$\big(\overbrace{2 \times (x \vee \mathtt{0x5c}) - (x \oplus \mathtt{0x5c})}^{\text{pattern matching}}\big) \times \mathtt{0x2} = \overbrace{(x + \mathtt{0x5c}) \times \mathtt{0x2}}^{\text{expansion}}$$

$$= 2x + \mathtt{0xb8}$$

Detailed example

$$\overbrace{(2 \times (x \vee \texttt{0x5c}) - (x \oplus \texttt{0x5c}))}^{\text{pattern matching}} \times \texttt{0x2} = \overbrace{(x + \texttt{0x5c}) \times \texttt{0x2}}^{\text{expansion}}$$
$$= 2x + \texttt{0xb8}$$

$$2 \times (x \vee \texttt{0x5c}) - (x \oplus \texttt{0x5c}) = 2 \times (x \vee \texttt{0x5c}) + (\neg(x \oplus \texttt{0x5c})) + 1$$

Detailed example

$$\begin{aligned}\overbrace{(2 \times (x \vee \mathtt{0x5c}) - (x \oplus \mathtt{0x5c}))}^{\text{pattern matching}} \times \mathtt{0x2} &= \overbrace{(x + \mathtt{0x5c}) \times \mathtt{0x2}}^{\text{expansion}} \\ &= 2x + \mathtt{0xb8}\end{aligned}$$

$$\begin{aligned}2 \times (x \vee \mathtt{0x5c}) - (x \oplus \mathtt{0x5c}) &= 2 \times (x \vee \mathtt{0x5c}) + (\neg(x \oplus \mathtt{0x5c})) + 1 \\ &= 2 \times (x \vee \mathtt{0x5c}) + (\neg x \oplus \mathtt{0x5c}) + 1\end{aligned}$$

Detailed example

$$\overbrace{(2 \times (x \vee \texttt{0x5c}) - (x \oplus \texttt{0x5c}))}^{\text{pattern matching}} \times \texttt{0x2} = \overbrace{(x + \texttt{0x5c}) \times \texttt{0x2}}^{\text{expansion}}$$
$$= 2x + \texttt{0xb8}$$

$$\begin{aligned}
2 \times (x \vee \texttt{0x5c}) - (x \oplus \texttt{0x5c}) &= 2 \times (x \vee \texttt{0x5c}) + (\neg(x \oplus \texttt{0x5c})) + 1 \\
&= 2 \times (x \vee \texttt{0x5c}) + (\neg x \oplus \texttt{0x5c}) + 1 \\
&= 2 \times (x \vee \texttt{0x5c}) + (x \oplus \texttt{0xa3}) + 1
\end{aligned}$$

Detailed example

$$\overbrace{(2 \times (x \vee \mathtt{0x5c}) - (x \oplus \mathtt{0x5c}))}^{\text{pattern matching}} \times \mathtt{0x2} = \overbrace{(x + \mathtt{0x5c}) \times \mathtt{0x2}}^{\text{expansion}}$$
$$= 2x + \mathtt{0xb8}$$

$$\begin{aligned}
2 \times (x \vee \mathtt{0x5c}) - (x \oplus \mathtt{0x5c}) &= 2 \times (x \vee \mathtt{0x5c}) + (\neg(x \oplus \mathtt{0x5c})) + 1 \\
&= 2 \times (x \vee \mathtt{0x5c}) + (\neg x \oplus \mathtt{0x5c}) + 1 \\
&= 2 \times (x \vee \mathtt{0x5c}) + (x \oplus \mathtt{0xa3}) + 1 \\
&= (2x \vee \mathtt{0xb8}) + (x \oplus \mathtt{0xa3}) + 1
\end{aligned}$$

Detailed example

$$\overbrace{(2 \times (x \vee \texttt{0x5c}) - (x \oplus \texttt{0x5c})}^{\text{pattern matching}}) \times \texttt{0x2} = \overbrace{(x + \texttt{0x5c}) \times \texttt{0x2}}^{\text{expansion}}$$
$$= 2x + \texttt{0xb8}$$

$$\begin{aligned}
2 \times (x \vee \texttt{0x5c}) - (x \oplus \texttt{0x5c}) &= 2 \times (x \vee \texttt{0x5c}) + (\neg(x \oplus \texttt{0x5c})) + 1 \\
&= 2 \times (x \vee \texttt{0x5c}) + (\neg x \oplus \texttt{0x5c}) + 1 \\
&= 2 \times (x \vee \texttt{0x5c}) + (x \oplus \texttt{0xa3}) + 1 \\
&= (2x \vee \texttt{0xb8}) + (x \oplus \texttt{0xa3}) + 1
\end{aligned}$$

Issues

- Obfuscation patterns must be known
- Detection of patterns is not trivial
- Properties of the set of rules: termination, confluence, convergence...

Symbolic Simplification with PAttern Matching (SSPAM)

- Implemented in Python, working with the `ast` module
- Contains its own pattern matcher
- Arithmetic simplification handled by the `sympy` module

Flexible matching

Query the SMT solver Z3 to prove equivalence of patterns. For example, if the known pattern is $2 \times (x \vee y) - (x \oplus y)$:

```
x = z3.BitVec('x', 8)
pattern = 2*(x | 0x5c) - (x ^ 0x5c)
expr = (2*x | 0xb8) + (x ^ 0xa3) + 1
z3.prove(pattern == expr)
```

Evaluation

- Simplifies all public examples of MBA obfuscated expressions

Evaluation

- Simplifies all public examples of MBA obfuscated expressions
- Evaluated on our own obfuscated samples:
  - On different expressions: $(x + y)$, $(x \oplus y)$, $(x \wedge 78)$, $(x \wedge 12)$
  - 50 obfuscated samples per expression
  - One pattern to rewrite each operator: $+, \oplus, \wedge, \vee$

Evaluation

- Simplifies all public examples of MBA obfuscated expressions
- Evaluated on our own obfuscated samples:
  - On different expressions: $(x + y)$, $(x \oplus y)$, $(x \wedge 78)$, $(x \wedge 12)$
  - 50 obfuscated samples per expression
  - One pattern to rewrite each operator: $+, \oplus, \wedge, \vee$
- Size of nodes reduced by approximatively 50%

Evaluation

- Simplifies all public examples of MBA obfuscated expressions
- Evaluated on our own obfuscated samples:
  - On different expressions: $(x + y)$, $(x \oplus y)$, $(x \wedge 78)$, $(x \wedge 12)$
  - 50 obfuscated samples per expression
  - One pattern to rewrite each operator: $+, \oplus, \wedge, \vee$
- Size of nodes reduced by approximatively 50%
- For two steps of obfuscation, around half of the expressions are *fully simplified*

Table of Contents

Background

Utility of MBA in Obfuscation

Our Contribution: MBA Simplification

Conclusion

Conclusion

- Definition of simplicity is not trivial
- Provided metrics to help characterize it
- Implemented an algorithm to simplify MBA obfuscated expressions

Future work

- Definition of simplicity applied to MBA expressions
- *Bit-vector size* metric
- Properties of the set of rewrite rules
- Improving SSPAM (strategies, bitwise simplification...)

Thank you!

https://github.com/quarkslab/sspam/

Improving MBA-based Obfuscation

- Expression-specific patterns: obfuscate $(x + 3)$ instead of $(x + y)$
- Conditionnal rewritings: equivalence for a certain range on variables
- More complex encodings: permutation polynomials modulo 2^n