

# Code obfuscation through Mixed Boolean-Arithmetic expressions

Arnau Gàmez i Montolio

---





[https://github.com/arnaugamez/talks/raw/master/2022/00\\_h-c0n/slides.pdf](https://github.com/arnaugamez/talks/raw/master/2022/00_h-c0n/slides.pdf)

# About



Hacker, Reverse Engineer & Mathematician.

# About



Hacker, Reverse Engineer & Mathematician.

Senior Malware Reverse Engineer @ Panda (WatchGuard)

# About



Hacker, Reverse Engineer & Mathematician.

Senior Malware Reverse Engineer @ Panda (WatchGuard)

Founder, Security Researcher & Trainer @ Fura Labs

# About



Hacker, Reverse Engineer & Mathematician.

Senior Malware Reverse Engineer @ Panda (WatchGuard)

Founder, Security Researcher & Trainer @ Fura Labs



 @arnaugamez



 @FuraLabs



This presentation may contain traces of maths and assembly

# Agenda



1. Code obfuscation
2. Preliminary MBA concepts
  - Introduction and Motivation
  - Obfuscation vs Cryptography
  - Definitions
    - Polynomial MBA expressions
    - Linear MBA expressions
3. Obfuscation with MBA expressions
  - MBA rewriting
  - Insertion of identities
  - Opaque constants





# Code obfuscation

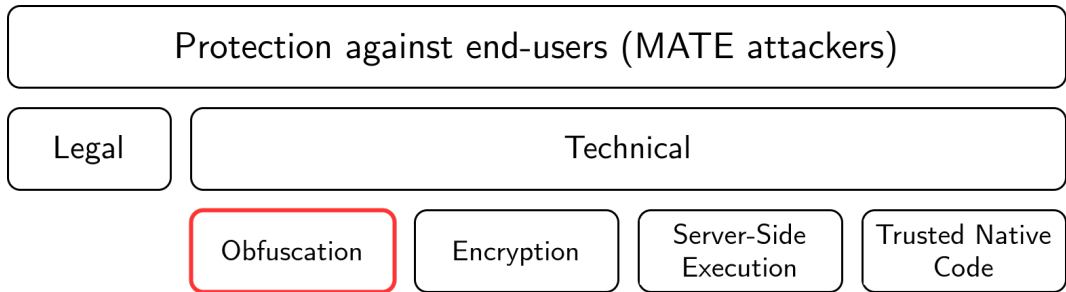
# Context



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.



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.





**Code obfuscation** is the process of transforming an input program  $P$  into a functionally equivalent program  $P'$  which is harder to analyze and to extract information that from  $P$ .

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



**Code obfuscation** is the process of transforming an input program  $P$  into a functionally equivalent program  $P'$  which is harder to analyze and to extract information that from  $P$ .

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

**Motivation:** ~~prevent~~ *complicate reverse engineering.*

# Presence



Software protection:

# Presence



Software protection:  
Intellectual property



Software protection:

- Intellectual property

- Digital Rights Management (DRM)





Software protection:

- Intellectual property

- Digital Rights Management (DRM)

- Anti-cheating



Software protection:

- Intellectual property

- Digital Rights Management (DRM)

- Anti-cheating

Malware threats:



Software protection:

- Intellectual property

- Digital Rights Management (DRM)

- Anti-cheating

Malware threats:

- Avoid automatic signature detection



Software protection:

- Intellectual property

- Digital Rights Management (DRM)

- Anti-cheating

Malware threats:

- Avoid automatic signature detection

- Slow down analysis → time → money



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 or instruction).



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 or instruction).

***Remark:** many “weak” techniques can be combined to create a “hard” obfuscation transformation.*



# Preliminary MBA concepts



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





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**.



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**.

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.

# Obfuscation vs Cryptography



In **cryptography**, the MBA expression is the direct result of the algorithm description. The resulting cryptosystem has to verify a set of properties (e.g. non-linearity, high algebraic degree) from a *black-box* point of view.

# Obfuscation vs Cryptography



In **cryptography**, the MBA expression is the direct result of the algorithm description. The resulting cryptosystem has to verify a set of properties (e.g. non-linearity, high algebraic degree) from a *black-box* point of view.

The complex form of writing is directly related to some kind of intrinsic computational (semantic) complexity for the resulting function: one wants the inverse computation to be difficult to deduce (without knowing the key).

# Obfuscation vs Cryptography



In **obfuscation**, the MBA expression is the result of rewriting iterations from a simpler expression which can have very simple *black-box* characteristics.

# Obfuscation vs Cryptography



In **obfuscation**, the MBA expression is the result of rewriting iterations from a simpler expression which can have very simple *black-box* characteristics.

There is no direct relation between the complex form of writing and any intrinsic computational (semantic) complexity of the resulting expression.

# Obfuscation vs Cryptography



In **obfuscation**, the MBA expression is the result of rewriting iterations from a simpler expression which can have very simple *black-box* characteristics.

There is no direct relation between the complex form of writing and any intrinsic computational (semantic) complexity of the resulting expression.

On the contrary, when obfuscating simple expressions, one knows that the complex form of writing is related to a semantically simpler expression.





We will be focusing on MBA expressions  
in the context of code (de)obfuscation

# Definitions



We choose to define MBA expressions by **explicitly describing the different building blocks** (operators) that compose them and how they are bundled together.

# Definitions



We choose to define MBA expressions by **explicitly describing the different building blocks** (operators) that compose them and how they are bundled together.

Two main categorizations are considered and studied in literature:



We choose to define MBA expressions by **explicitly describing the different building blocks** (operators) that compose them and how they are bundled together.

Two main categorizations are considered and studied in literature:

Linear MBA expressions



We choose to define MBA expressions by **explicitly describing the different building blocks** (operators) that compose them and how they are bundled together.

Two main categorizations are considered and studied in literature:

Linear MBA expressions  $\subset$



We choose to define MBA expressions by **explicitly describing the different building blocks** (operators) that compose them and how they are bundled together.

Two main categorizations are considered and studied in literature:

Linear MBA expressions  $\subset$  Polynomial MBA expressions

# Polynomial MBA expressions



A polynomial MBA expression consists of a **sum of terms**, each one composed by an  $n$ -bit constant  $a_i$  times the **product** of several **bitwise expressions** on a number  $t$  of  $n$ -bit variables.

# Polynomial MBA expressions



A polynomial MBA expression consists of a **sum of terms**, each one composed by an  **$n$ -bit constant**  $a_i$  times the **product** of several **bitwise expressions** on **a number  $t$  of  $n$ -bit variables**.

$$E = \sum_{i \in I} a_i \cdot \left( \prod_{j \in J_i} e_{i,j}(x_1, \dots, x_t) \right)$$



# Polynomial MBA expressions



A polynomial MBA expression consists of a **sum of terms**, each one composed by an  **$n$ -bit constant**  $a_i$  times the **product** of several **bitwise expressions** on **a number  $t$  of  $n$ -bit variables**.

$$E = \sum_{i \in I} a_i \cdot \left( \prod_{j \in J_i} e_{i,j}(x_1, \dots, x_t) \right)$$

Example

$$E = \underline{\underline{43}} \left( \underline{\underline{x \wedge y \vee z}} \right)^2 \left( \underline{\underline{(x \oplus y) \wedge z \vee t}} \right) + \underline{\underline{2x}} + \underline{\underline{123}} \left( \underline{\underline{x \vee y}} \right) z t^2$$

# Linear MBA expressions



A polynomial MBA expression of the form

$$E = \sum_{i \in I} a_i \cdot e_i(x_1, \dots, x_t)$$

is called a linear MBA expression.

# Linear MBA expressions



A polynomial MBA expression of the form

$$E = \sum_{i \in I} a_i \cdot e_i(x_1, \dots, x_t)$$

is called a linear MBA expression.

They are defined by imposing **just one bitwise expression for each term** instead of a product of an arbitrary number of them.

# Linear MBA expressions



A polynomial MBA expression of the form

$$E = \sum_{i \in I} a_i \cdot e_i(x_1, \dots, x_t)$$

is called a linear MBA expression.

They are defined by imposing **just one bitwise expression for each term** instead of a product of an arbitrary number of them.

***Note:*** In practice, you can vaguely think of linearity as a restriction not allowing variables to end up being multiplied together.

# Linear MBA expressions



$$E = \sum_{i \in I} a_i \cdot e_i(x_1, \dots, x_t)$$

Example

$$E = \underbrace{(x \oplus y)}_{\text{red}} + 2 \underbrace{(x \wedge y)}_{\text{blue}}$$

# Linear MBA expressions



Notice that, assuming variables of the same *bit size*, the previous MBA expression example  $E = (x \oplus y) + 2(x \wedge y)$  simplifies to  $E_{simp} = x + y$ .

# Linear MBA expressions



Notice that, assuming variables of the same *bit size*, the previous MBA expression example  $E = (x \oplus y) + 2(x \wedge y)$  simplifies to  $E_{simp} = x + y$ .

Namely,  $E$  is a more complex expression than  $E_{simp}$  syntactically speaking, but they are semantically equivalent.

# Linear MBA expressions



We can easily verify this equivalence with an SMT solver like Z3.



# Linear MBA expressions



We can easily verify this equivalence with an SMT solver like Z3.

---

```
from z3 import *
x = BitVec('x', 8)
y = BitVec('y', 8)
E = (x ^ y) + 2 * (x & y)    #  $E = (x \oplus y) + 2(x \wedge y)$ 
E_simp = x + y              #  $E_{simp} = x + y$ 
prove (E == E_simp)         #  $E \stackrel{?}{=} E_{simp}$ 
```

---

# Linear MBA expressions



We can easily verify this equivalence with an SMT solver like Z3.

---

```
from z3 import *
x = BitVec('x', 8)
y = BitVec('y', 8)
E = (x ^ y) + 2 * (x & y)    #  $E = (x \oplus y) + 2(x \wedge y)$ 
E_simp = x + y              #  $E_{simp} = x + y$ 
prove (E == E_simp)         #  $E \stackrel{?}{=} E_{simp}$ 
```

---

```
$ python eq.py
proved
```

---



# Obfuscation with MBA expressions

# Obfuscation idea



The previous example already suggests a basic obfuscation idea: we could replace the arithmetic sum  $+$  of two variables in our code by the more complex expression involving  $\oplus$  and  $\wedge$  boolean operators, while preserving the code semantics.

# Obfuscation idea



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  $E_1$ .

# Obfuscation idea



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  $E_1$ .

The process relies on two differentiated components, which can be used either alone or combined:

# Obfuscation idea



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  $E_1$ .

The process relies on two differentiated components, which can be used either alone or combined:

MBA rewriting & Insertion of identities



A chosen operator is rewritten with an equivalent MBA expression.





A chosen operator is rewritten with an equivalent MBA expression.

Example

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

# 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 on  $n$ -bits.

# 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 on  $n$ -bits.

The function  $f$  is often an affine function.

# 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 on  $n$ -bits.

The function  $f$  is often an affine function.

**Note:** For our usage, you can vaguely think of affine functions as those with the form  $f(e) = a \cdot e + b$ , where  $a, b$  are  $n$ -bit constants and  $e$  is our MBA subexpression.

# Insertion of identities



## Example

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

$$f : x \mapsto 39x + 23 \qquad f^{-1} : x \mapsto 151x + 111$$

# Insertion of identities



## Example

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

$$f : x \mapsto 39x + 23 \quad f^{-1} : x \mapsto 151x + 111$$

Consider now  $E_2$  obtained by applying the previous rewriting rule to  $E_1$ :

$$E_2 = (x \oplus y) + 2 \times (x \wedge y)$$

# Insertion of identities



## Example

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

$$f : x \mapsto 39x + 23 \quad f^{-1} : x \mapsto 151x + 111$$

Consider now  $E_2$  obtained by applying the previous rewriting rule to  $E_1$ :

$$E_2 = (x \oplus y) + 2 \times (x \wedge y)$$

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

$$E_{tmp} = f(E_2) = 39 \times E_2 + 23$$

$$E_3 = f^{-1}(E_{tmp}) = 151 \times E_{tmp} + 111$$

# Insertion of identities



## Example

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

$$f : x \mapsto 39x + 23 \quad f^{-1} : x \mapsto 151x + 111$$

Consider now  $E_2$  obtained by applying the previous rewriting rule to  $E_1$ :

$$E_2 = (x \oplus y) + 2 \times (x \wedge y)$$

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

$$E_{tmp} = f(E_2) = 39 \times E_2 + 23$$

$$E_3 = f^{-1}(E_{tmp}) = 151 \times E_{tmp} + 111$$

Finally, expand  $E_3$  to observe the final obfuscated expression:

$$E_3 = 151 \times (39 \times ((x \oplus y) + 2 \times (x \wedge y)) + 23) + 111$$



# Practical demo



Let:

$$E_1 = x + y$$

$$E_2 = (x \oplus y) + 2 \times (x \wedge y)$$

$$E_3 = 151 \times (39 \times ((x \oplus y) + 2 \times (x \wedge y)) + 23) + 111$$

# Practical demo



Let:

$$E_1 = x + y$$

$$E_2 = (x \oplus y) + 2 \times (x \wedge y)$$

$$E_3 = 151 \times (39 \times ((x \oplus y) + 2 \times (x \wedge y)) + 23) + 111$$

Task:

**1** Compare syntactic complexity of  $E_1$ ,  $E_2$  and  $E_3$ .

# Practical demo



Let:

$$E_1 = x + y$$

$$E_2 = (x \oplus y) + 2 \times (x \wedge y)$$

$$E_3 = 151 \times (39 \times ((x \oplus y) + 2 \times (x \wedge y)) + 23) + 111$$

Task:

- 1 Compare syntactic complexity of  $E_1$ ,  $E_2$  and  $E_3$ .
- 2 Observe semantic equivalence of  $E_1$ ,  $E_2$  and  $E_3$ .

# Practical demo



Let:

$$E_1 = x + y$$

$$E_2 = (x \oplus y) + 2 \times (x \wedge y)$$

$$E_3 = 151 \times (39 \times ((x \oplus y) + 2 \times (x \wedge y)) + 23) + 111$$

Task:

- 1 Compare syntactic complexity of  $E_1$ ,  $E_2$  and  $E_3$ .
- 2 Observe semantic equivalence of  $E_1$ ,  $E_2$  and  $E_3$ .
- 3 Prove semantic equivalence of  $E_1$ ,  $E_2$  and  $E_3$ .

# Practical demo



Task 1: Compare syntactic complexity of  $E_1$ ,  $E_2$  and  $E_3$ .

# Practical demo



Task 1: Compare syntactic complexity of  $E_1$ ,  $E_2$  and  $E_3$ .

---

```
uint8_t E1(uint8_t x, uint8_t y)
{
    return x+y;
}
```

---

---

```
movzx edx, byte [var_4h]
movzx eax, byte [var_8h]
add     eax, edx
```

---

# Practical demo



Task 1: Compare syntactic complexity of  $E_1$ ,  $E_2$  and  $E_3$ .

---

```
uint8_t E1(uint8_t x, uint8_t y)
{
    return x+y;
}
```

---

---

```
movzx edx, byte [var_4h]
movzx eax, byte [var_8h]
add    eax, edx
```

---

---

```
uint8_t E2(uint8_t x, uint8_t y)
{
    return (x^y)+2*(x&y);
}
```

---

---

```
movzx eax, byte [var_4h]
xor    al, byte [var_8h]
mov    edx, eax
movzx  eax, byte [var_4h]
and    al, byte [var_8h]
add    eax, eax
add    eax, edx
```

---

# Practical demo



Task 1: Compare syntactic complexity of  $E_1$ ,  $E_2$  and  $E_3$ .

```
uint8_t E1(uint8_t x, uint8_t y)
{
    return x+y;
}
```

```
movzx edx, byte [var_4h]
movzx eax, byte [var_8h]
add  eax, edx
```

```
uint8_t E2(uint8_t x, uint8_t y)
{
    return (x^y)+2*(x&y);
}
```

```
movzx eax, byte [var_4h]
xor  al, byte [var_8h]
mov  edx, eax
movzx eax, byte [var_4h]
and  al, byte [var_8h]
add  eax, eax
add  eax, edx
```

```
uint8_t E3(uint8_t x, uint8_t y)
{
    return 151*(39*((x^y)+2*(x&y))+23)+111;
}
```

```
movzx eax, byte [var_4h]
xor  al, byte [var_8h]
movzx edx, al
movzx eax, byte [var_4h]
and  al, byte [var_8h]
movzx eax, al
add  eax, eax
add  eax, edx
imul eax, eax, 0x27
add  eax, 0x17
mov  edx, 0xffffffff97
imul eax, edx
add  eax, 0x6f
```



# Practical demo



Task 2: Observe semantic equivalence of  $E_1$ ,  $E_2$  and  $E_3$ .

# Practical demo



Task 2: Observe semantic equivalence of  $E_1$ ,  $E_2$  and  $E_3$ .

---

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

uint8_t E1(uint8_t x, uint8_t y)
{ return x + y; }
uint8_t E2(uint8_t x, uint8_t y)
{ return (x ^ y) + 2 * (x & y); }
uint8_t E3(uint8_t x, uint8_t y)
{ return 151 * (39 * ((x ^ y) + 2 * (x & y)) + 23) + 111; }

int main(int argc, char* argv[])
{
    uint8_t x = (uint8_t) atoi (argv[1]);
    uint8_t y = (uint8_t) atoi (argv[2]);
    printf ("%s(%d, %d) = %d\n", "E1", x, y, E1(x, y));
    printf ("%s(%d, %d) = %d\n", "E2", x, y, E2(x, y));
    printf ("%s(%d, %d) = %d\n", "E3", x, y, E3(x, y));
    return 0;
}
```

---

# Practical demo



Task 2: Observe semantic equivalence of  $E_1$ ,  $E_2$  and  $E_3$ .

---

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

uint8_t E1(uint8_t x, uint8_t y)
{ return x + y; }
uint8_t E2(uint8_t x, uint8_t y)
{ return (x ^ y) + 2 * (x & y); }
uint8_t E3(uint8_t x, uint8_t y)
{ return 151 * (39 * ((x ^ y) + 2 * (x & y)) + 23) + 111; }

int main(int argc, char* argv[])
{
    uint8_t x = (uint8_t) atoi (argv[1]);
    uint8_t y = (uint8_t) atoi (argv[2]);
    printf ("%s(%d, %d) = %d\n", "E1", x, y, E1(x, y));
    printf ("%s(%d, %d) = %d\n", "E2", x, y, E2(x, y));
    printf ("%s(%d, %d) = %d\n", "E3", x, y, E3(x, y));
    return 0;
}
```

---

---

```
$ gcc linear_mba.c -o linear_mba
$ ./linear_mba 1 2
E1(1, 2) = 3
E2(1, 2) = 3
E3(1, 2) = 3
$ ./linear_mba 23 89
E1(23, 89) = 112
E2(23, 89) = 112
E3(23, 89) = 112
```

---

# Practical demo

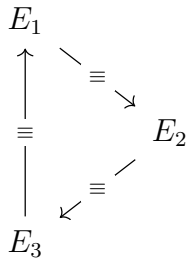


Task 3: Prove semantic equivalence of  $E_1$ ,  $E_2$  and  $E_3$ .

# Practical demo



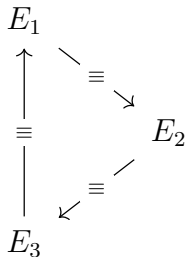
Task 3: Prove semantic equivalence of  $E_1$ ,  $E_2$  and  $E_3$ .



# Practical demo



Task 3: Prove semantic equivalence of  $E_1$ ,  $E_2$  and  $E_3$ .



---

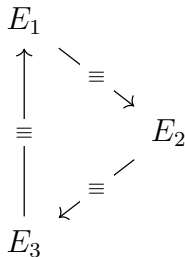
```
from z3 import *  
x = BitVec('x', 8)  
y = BitVec('y', 8)  
  
E1 = x + y  
E2 = (x ^ y) + 2 * (x & y)  
E3 = 151 * (39 * ((x ^ y) + 2 * (x & y)) + 23) + 111  
  
prove (E1 == E2)  
prove (E2 == E3)  
prove (E3 == E1)
```

---

# Practical demo



Task 3: Prove semantic equivalence of  $E_1$ ,  $E_2$  and  $E_3$ .



```
from z3 import *
x = BitVec('x', 8)
y = BitVec('y', 8)

E1 = x + y
E2 = (x ^ y) + 2 * (x & y)
E3 = 151 * (39 * ((x ^ y) + 2 * (x & y)) + 23) + 111

prove (E1 == E2)
prove (E2 == E3)
prove (E3 == E1)
```

```
$ python prove.py
proved
proved
proved
```

# Opaque constants



This technique allows to hide a target constant  $K$ .



# Opaque constants



This technique allows to hide a target constant  $K$ .

It combines the power of MBA expressions with *permutation polynomials*.

# Opaque constants



This technique allows to hide a target constant  $K$ .

It combines the power of MBA expressions with *permutation polynomials*.

A permutation polynomial is a polynomial that acts as a permutation of the elements of the set they apply to (in our case,  $n$ -bit values), i.e. they define a 1-to-1 map (bijection).

# Opaque constants



This technique allows to hide a target constant  $K$ .

It combines the power of MBA expressions with *permutation polynomials*.

A permutation polynomial is a polynomial that acts as a permutation of the elements of the set they apply to (in our case,  $n$ -bit values), i.e. they define a 1-to-1 map (bijection).

Thus, for any permutation polynomial  $P$ , there exists another one  $Q$  that defines the inverse map, i.e., for all  $n$ -bit  $X$  values we have that:

$$P(Q(X)) = X$$

# Opaque constants



Let:

$K$  be an  $n$ -bit target constant to hide,

# Opaque constants



Let:

$K$  be an  $n$ -bit target constant to hide,

$P$  and  $Q$  polynomials with  $n$ -bit coefficients and acting as inverse 1-to-1 maps, i.e.  $P(Q(X)) = X$  for all  $X$ ,

# Opaque constants



Let:

$K$  be an  $n$ -bit target constant to hide,

$P$  and  $Q$  polynomials with  $n$ -bit coefficients and acting as inverse 1-to-1 maps, i.e.  $P(Q(X)) = X$  for all  $X$ ,

$E$  be an MBA expression of  $n$ -bit variables non-trivially equal to zero, i.e.  $E(x_1, \dots, x_t) = 0$  for any input variables  $x_1, \dots, x_t$ .

# Opaque constants



Let:

$K$  be an  $n$ -bit target constant to hide,

$P$  and  $Q$  polynomials with  $n$ -bit coefficients and acting as inverse 1-to-1 maps, i.e.  $P(Q(X)) = X$  for all  $X$ ,

$E$  be an MBA expression of  $n$ -bit variables non-trivially equal to zero, i.e.  $E(x_1, \dots, x_t) = 0$  for any input variables  $x_1, \dots, x_t$ .

Then, the constant  $K$  can be replaced by  $P(E + Q(K))$  for any values taken by  $(x_1, \dots, x_t)$ .

# Practical demo



Working on 8-bit values, let:

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

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

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



# Practical demo



Working on 8-bit values, let:

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

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

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

Task:

**1** Check that  $P$  and  $Q$  define inverse maps, i.e.  $P(Q(X)) = X$  for all  $X$ .

# Practical demo



Working on 8-bit values, let:

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

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

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

Task:

- 1 Check that  $P$  and  $Q$  define inverse maps, i.e.  $P(Q(X)) = X$  for all  $X$ .
- 2 Check that  $E$  defines a non-trivially equal to zero MBA expression, i.e.  $E(x, y) = 0$  for all  $x, y$ .

# Practical demo



Working on 8-bit values, let:

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

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

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

Task:

- 1 Check that  $P$  and  $Q$  define inverse maps, i.e.  $P(Q(X)) = X$  for all  $X$ .
- 2 Check that  $E$  defines a non-trivially equal to zero MBA expression, i.e.  $E(x, y) = 0$  for all  $x, y$ .
- 3 Create an opaque constant function using  $P$ ,  $Q$  and  $E$  to hide the constant  $K = 123$ .

# Practical demo



Working on 8-bit values, let:

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

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

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

Task:

- 1 Check that  $P$  and  $Q$  define inverse maps, i.e.  $P(Q(X)) = X$  for all  $X$ .
- 2 Check that  $E$  defines a non-trivially equal to zero MBA expression, i.e.  $E(x, y) = 0$  for all  $x, y$ .
- 3 Create an opaque constant function using  $P$ ,  $Q$  and  $E$  to hide the constant  $K = 123$ .
- 4 Check the previous opaque constant.

# Practical demo



Task 1: Check that  $P$  and  $Q$  define inverse maps (bruteforce).

# Practical demo



Task 1: Check that  $P$  and  $Q$  define inverse maps (bruteforce).

---

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

uint8_t P(uint8_t x) { return 97*x + 248*x*x; }
uint8_t Q(uint8_t x) { return 161*x + 136*x*x; }

int main(int argc, char* argv[])
{
    uint8_t i = 0;
    do
    {
        if (P(Q(i)) != i) { printf("P(Q(X)) != X\n"); return -1; }
        i++;
    } while (i != 0);
    printf("P(Q(X)) = X\n");
    return 0;
}
```

---

# Practical demo



Task 1: Check that  $P$  and  $Q$  define inverse maps (bruteforce).

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

uint8_t P(uint8_t x) { return 97*x + 248*x*x; }
uint8_t Q(uint8_t x) { return 161*x + 136*x*x; }

int main(int argc, char* argv[])
{
    uint8_t i = 0;
    do
    {
        if (P(Q(i)) != i) { printf("P(Q(X)) != X\n"); return -1; }
        i++;
    } while (i != 0);
    printf("P(Q(X)) = X\n");
    return 0;
}
```

```
$ gcc check_poly.c -o check_poly
$ ./check_poly
P(Q(X)) = X
```

# Practical demo



Task 1: Check that  $P$  and  $Q$  define inverse maps (SMT).





Task 1: Check that  $P$  and  $Q$  define inverse maps (SMT).

---

```
from z3 import *  
X = BitVec('X', 8)  
  
def P(X): return 97*X + 248*X*X  
def Q(X): return 161*X + 136*X*X  
  
prove(P(Q(X)) == X)
```

---



Task 1: Check that  $P$  and  $Q$  define inverse maps (SMT).

---

```
from z3 import *  
X = BitVec('X', 8)  
  
def P(X): return 97*X + 248*X*X  
def Q(X): return 161*X + 136*X*X  
  
prove(P(Q(X)) == X)
```

---

```
$ python check_poly.py  
proved
```

---

# Practical demo



Task 2: Check that  $E$  is non-trivially equal to zero (bruteforce).

# Practical demo



Task 2: Check that  $E$  is non-trivially equal to zero (bruteforce).

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

uint8_t E(uint8_t x, uint8_t y) { return x-y + 2*(~x&y) - (x^y); }

int main(int argc, char* argv[])
{
    uint8_t i = 0; uint8_t j = 0;
    do
    {
        do
        {
            if (E(i, j) != 0) { printf("E(x, y) != 0\n"); return -1; }
            j++;
        } while (j != 0);
        i++;
    } while (i != 0);
    printf("E(x, y) = 0\n"); return 0;
}
```

# Practical demo



Task 2: Check that  $E$  is non-trivially equal to zero (bruteforce).

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
```

```
uint8_t E(uint8_t x, uint8_t y) { return x-y + 2*(~x&y) - (x^y); }
```

```
int main(int argc, char* argv[])
{
    uint8_t i = 0; uint8_t j = 0;
    do
    {
        do
        {
            if (E(i, j) != 0) { printf("E(x, y) != 0\n"); return -1; }
            j++;
        } while (j != 0);
        i++;
    } while (i != 0);
    printf("E(x, y) = 0\n"); return 0;
}
```

```
$ gcc check_mba.c -o check_mba
$ ./check_mba
E(x, y) = 0
```

# Practical demo



Task 2: Check that  $E$  is non-trivially equal to zero (SMT).



Task 2: Check that  $E$  is non-trivially equal to zero (SMT).

---

```
from z3 import *
x = BitVec('x', 8)
y = BitVec('y', 8)

def E(x, y): return x-y + 2*(~x&y) - (x^y)

prove(E(x, y) == 0)
```

---



Task 2: Check that  $E$  is non-trivially equal to zero (SMT).

---

```
from z3 import *  
x = BitVec('x', 8)  
y = BitVec('y', 8)  
  
def E(x, y): return x-y + 2*(~x&y) - (x^y)  
  
prove(E(x, y) == 0)
```

---

```
$ python check_mba.py  
proved
```

---



# Practical demo



Task 3: Create an opaque constant function.

# Practical demo



## Task 3: Create an opaque constant function.

---

```
from z3 import *

X = BitVec('X', 8)
def P(X): return 97*X + 248*X*X
def Q(X): return 161*X + 136*X*X

x = BitVec('x', 8)
y = BitVec('y', 8)
def E(x, y): return x-y + 2*(~x&y) - (x^y)

K = BitVecVal(123, 8)

# Opaque Constant
OC = P(E(x,y) + Q(K))

# Apply basic simplification rules
print(simplify(OC))
```

---

# Practical demo



## Task 3: Create an opaque constant function.

```
from z3 import *

X = BitVec('X', 8)
def P(X): return 97*X + 248*X*X
def Q(X): return 161*X + 136*X*X

x = BitVec('x', 8)
y = BitVec('y', 8)
def E(x, y): return x-y + 2*(~x&y) - (x^y)

K = BitVecVal(123, 8)

# Opaque Constant
OC = P(E(x,y) + Q(K))

# Apply basic simplification rules
print (simplify(OC))
```

```
$ python create_oc.py
195 +
97*x +
159*y +
194*~(x | ~y) +
159*(x ^ y) +
(163 + x + 255*y + 2*~(x | ~y) + 255*(x ^ y))*
(232 + 248*x + 8*y + 240*~(x | ~y) + 8*(x ^ y))
```

# Practical demo



Task 4: Check an opaque constant function (bruteforce).

# Practical demo



## Task 4: Check an opaque constant function (bruteforce).

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

uint8_t OC(uint8_t x, uint8_t y)
{
    return 195 + 97*x + 159*y +
        194*~(x | ~y) + 159*(x ^ y) +
        (163 + x + 255*y + 2*~(x | ~y) + 255*(x ^ y))*
        (232 + 248*x + 8*y + 240*~(x | ~y) + 8*(x ^ y));
}

int main(int argc, char* argv[])
{
    uint8_t i = 0; uint8_t j = 0;
    do
    {
        /* ... */
    }
```

```
do
{
    if (OC(i, j) != 123)
    {
        printf("OC(x, y) != 123\n");
        return -1;
    }
    j++;
} while (j != 0);
i++;
} while (i != 0);
printf("OC(x, y) = 123\n"); return 0;
}
```

# Practical demo



## Task 4: Check an opaque constant function (bruteforce).

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

uint8_t OC(uint8_t x, uint8_t y)
{
    return 195 + 97*x + 159*y +
        194*~(x | ~y) + 159*(x ^ y) +
        (163 + x + 255*y + 2*~(x | ~y) + 255*(x ^ y))*
        (232 + 248*x + 8*y + 240*~(x | ~y) + 8*(x ^ y));
}

int main(int argc, char* argv[])
{
    uint8_t i = 0; uint8_t j = 0;
    do
    {
        /* ... */
    }
}
```

```
do
{
    if (OC(i, j) != 123)
    {
        printf("OC(x, y) != 123\n");
        return -1;
    }
    j++;
} while (j != 0);
i++;
} while (i != 0);
printf("OC(x, y) = 123\n"); return 0;
}
```

```
$ gcc check_oc.c -o check_oc
$ ./check_oc
OC(x, y) = 123
```

# Practical demo



Task 4: Check an opaque constant function (SMT).



## Task 4: Check an opaque constant function (SMT).

---

```
from z3 import *

x = BitVec('x', 8)
y = BitVec('y', 8)

def OC(x, y):
    return 195 + 97*x + 159*y +\
        194*~(x | ~y) + 159*(x ^ y) +\
        (163 + x + 255*y + 2*~(x | ~y) + 255*(x ^ y))*\
        (232 + 248*x + 8*y + 240*~(x | ~y) + 8*(x ^ y))

prove(OC(x, y) == 123)
```

---





## Task 4: Check an opaque constant function (SMT).

---

```
from z3 import *
```

```
x = BitVec('x', 8)
```

```
y = BitVec('y', 8)
```

```
def OC(x, y):
```

```
    return 195 + 97*x + 159*y +\  
    194*~(x | ~y) + 159*(x ^ y) +\  
    (163 + x + 255*y + 2*~(x | ~y) + 255*(x ^ y))*\  
    (232 + 248*x + 8*y + 240*~(x | ~y) + 8*(x ^ y))
```

```
prove(OC(x, y) == 123)
```

---

---

```
$ python check_oc.py  
proved
```

---

# Summary



We have seen how to apply several MBA obfuscation techniques: MBA rewriting, insertion of identities and opaque constants.

# Summary



We have seen how to apply several MBA obfuscation techniques: MBA rewriting, insertion of identities and opaque constants.

For this purpose, we have used: rewrite rules, affine functions, non-trivially equal to zero MBA expressions and permutation polynomials.

# What would come next?



Learn methods to generate:

# What would come next?



Learn methods to generate:

- Non-trivially equal to zero MBA expressions

# What would come next?



Learn methods to generate:

- Non-trivially equal to zero MBA expressions
- Linear MBA rewrite rules

# What would come next?



Learn methods to generate:

- Non-trivially equal to zero MBA expressions
- Linear MBA rewrite rules
- Pairs of inverse affine functions

# What would come next?



Learn methods to generate:

- Non-trivially equal to zero MBA expressions
- Linear MBA rewrite rules
- Pairs of inverse affine functions
- Pairs of inverse permutation polynomials





*Code deobfuscation by program synthesis-aided simplification of Mixed Boolean-Arithmetic expressions*<sup>1</sup> – Arnau Gàmez i Montolio

*Obfuscation with Mixed Boolean Arithmetic Expressions*<sup>2</sup> – Ninon Eyrolles

*Information Hiding in Software with Mixed Boolean-Arithmetic Transforms*<sup>3</sup>  
– Y. Zhou et al.

---

<sup>1</sup><https://github.com/arnaugamez/tfg>

<sup>2</sup><https://tel.archives-ouvertes.fr/tel-01623849/document>

<sup>3</sup>[https://link.springer.com/chapter/10.1007/978-3-540-77535-5\\_5](https://link.springer.com/chapter/10.1007/978-3-540-77535-5_5)




[https://github.com/arnaugamez/talks/tree/master/2022/00\\_h-c0n](https://github.com/arnaugamez/talks/tree/master/2022/00_h-c0n)



## To know more


---

 [blog.furalabs.com](https://blog.furalabs.com)

 [furalabs.com/trainings](https://furalabs.com/trainings)

## Get in touch

---

 [@arnaugamez](https://twitter.com/arnaugamez)

 [@FuraLabs](https://twitter.com/FuraLabs)

EOF