

# **Fun with Binary Polynomials**

## **Obfuscation and Reverse Engineering**

**Arnau Gàmez i Montolio**

TyphoonCon 2025 - May 30, 2025 - Seoul

# Agenda

- Introduction
- Binary polynomials
- Obfuscation
- Reverse engineering and analysis
- Conclusions

# 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

# Disclaimer

Not representing my employer or university.

# Introduction

## Motivation

1. Design new *obfuscation primitives* to raise the bar of current software protection transformations.
2. Explore formal constructions to reason about code (transformations) and arbitrary bit-vector mappings in an algebraically friendly way.
3. **Have fun doing math.**

# 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.

## Data-flow

Computational process through which a variable (register, memory location) ends up holding a certain value at a given point in the program execution.

## Control-flow

Order in which individual statements (instructions) or functions are executed. Guided by structures like conditional branching, loops, function calls...

## 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.

# Binary polynomials

*Preliminary*

# Modular arithmetic

We work on  $w$ -bit values, which can be seen as integers modulo  $2^w$ .

$$\{0, 1\}^w \cong \mathbb{Z}_{2^w}$$

In other words, values *wrap around* (overflow) at  $2^w$ .



$$251 + 9 = 260 \equiv 4 \pmod{2^8}$$

$$10 - 47 = -37 \equiv 219 \pmod{2^8}$$

$$36 \cdot 17 = 612 \equiv 100 \pmod{2^8}$$

$$123^{-1} \equiv 179 \pmod{2^8}$$

# Mixed Boolean-Arithmetic (MBA) expressions

Algebraic expressions on some bitsize 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>

## Obfuscate operators / expressions

$x + y$

=====

```
(((((0x62 + (0xBF * ((0x22 + (0xFF * ((x | (0xFA ^ x)) & (y | 0x05)))) +
((0xD0 * ~(~((y ^ ~x) | 0xFA) ^ ~y)) * ~(((0x05 | (y ^ x)) ^ x) ^
0xFA)))))) + ((0xD0 * ((0x22 + (0xFF * ((0xFA | (x & x)) & (0x05 | y)))) +
((0xD0 * ((y | (0x05 | y)) & (x | 0xFA))) * ~(((0x05 | (y ^ x)) ^ x) ^
0xFA)))) * ((0x22 + (0xFF * ((y & (y & 0xFA)) | (x & 0x05)))) + ((0xD0 *
((0xFA & (y & y)) | (0x05 & x))) * ((y & (0xFA & y)) ^ (0x05 & x)))))) +
0x5E) + (0xEF * ((0xBE + (0x4F * ~((0xFA | y) ^ (x | (0x05 | x)))))) +
((0x10 * ~(~((y ^ ~x) | 0xFA) ^ ~x)) * ~(((y ^ (0xFA ^ x)) | 0x05) ^
y)))))) + ((0x10 * ((0xBE + (0x4F * ((x & (0xFA & x)) ^ (y & 0x05)))) +
((0x10 * ((y | (y ^ 0xFA)) & (x | 0x05))) * ((0xFA & (x & x)) ^ (0x05 &
y)))))) * ((0xBE + (0x4F * ~(((0xFA | (y ^ x)) ^ x) ^ 0x05))) + ((0x10 *
((0x05 & (x ^ y)) ^ (x | x))) * ((y & (y & 0x05)) ^ (0xFA & x))))))
```

## Obfuscate constants as runtime computations

123

===

```
((0xA3 + (0x81 * (((0x73 * ~(~y | 0x02) & (0xFD & (~x ^ y)))) + (0x73 *
~(((y ^ (x | x)) | 0x02) | y))) + 0x4B))) + ((0x60 * (((0x73 * ((x | (x ^
0x02)) | (x ^ y))) + (0x73 * ~(~y & x) | (0x02 | (~y ^ 0xFD)))) + 0x4B))
* (((0x73 * ((y | (x ^ y)) | (0x02 | y))) + (0x73 * ~(~y & 0x02) | (y |
(0x02 | x)))))) + 0x4B)))
```

# Polynomial

A mathematical expression that consists of a sum of terms. Each term is the product of a coefficient and one or more variables, possibly raised to a non-negative power.

$$P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

$$P(x, y) = a_{00} + a_{10}x + a_{01}y + a_{11}xy + a_{20}x^2 + a_{02}y^2 + \dots + a_{ij}x^i y^j$$

...

# Binary polynomials

## *Basics*

# Binary polynomial

A polynomial over values of  $w$  bits, i.e., modulo  $2^w$ .

$$P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n \pmod{2^w}$$

$$P(x) = 5x^4 + 250x^3 + 248x^2 + 252x + 249 \pmod{2^8}$$

$$\begin{aligned} P(12) &= 249 + 252 \cdot 12 + 248 \cdot 12^2 + 250 \cdot 12^3 + 5 \cdot 12^4 \\ &= 574665 \\ &\equiv 201 \pmod{2^8} \end{aligned}$$



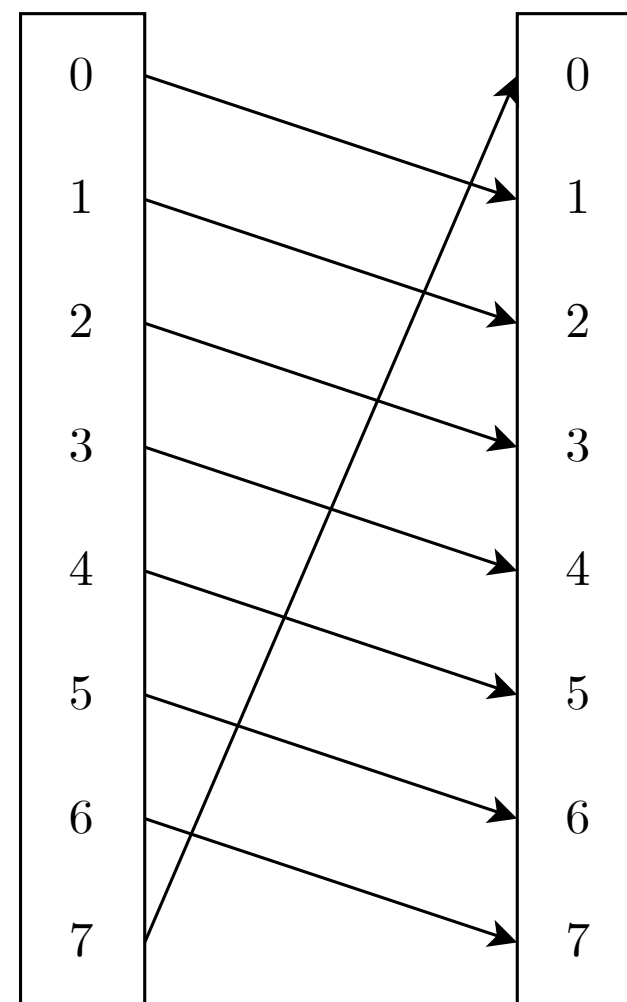
# Polynomial *function*

A function (a mapping) that can be defined by evaluating a polynomial (expression).

The *polynomial* is the (written) expression.

$$P(x) = 1 + 5x + 4x^2 \pmod{2^3}.$$

The *polynomial function* is the actual mapping the polynomial describes when evaluated.



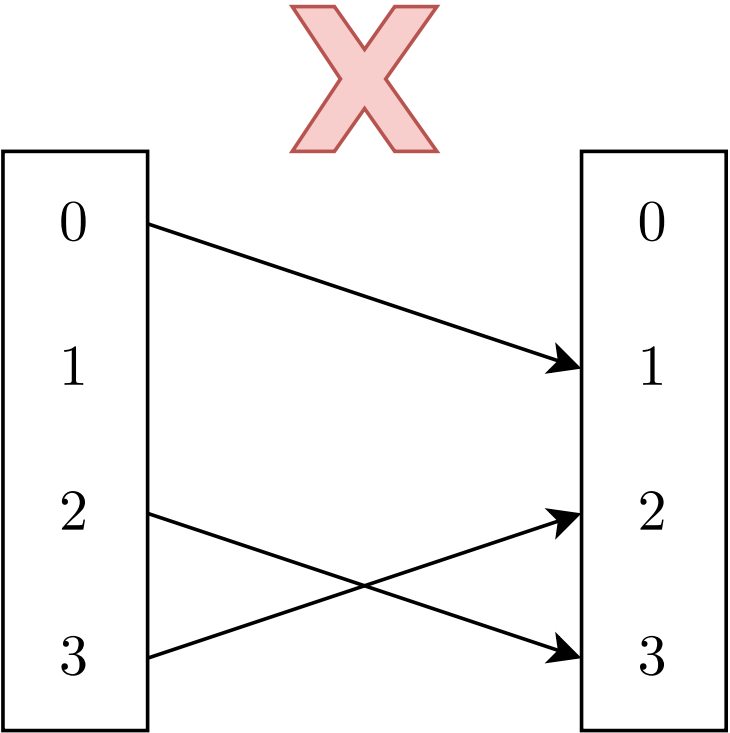
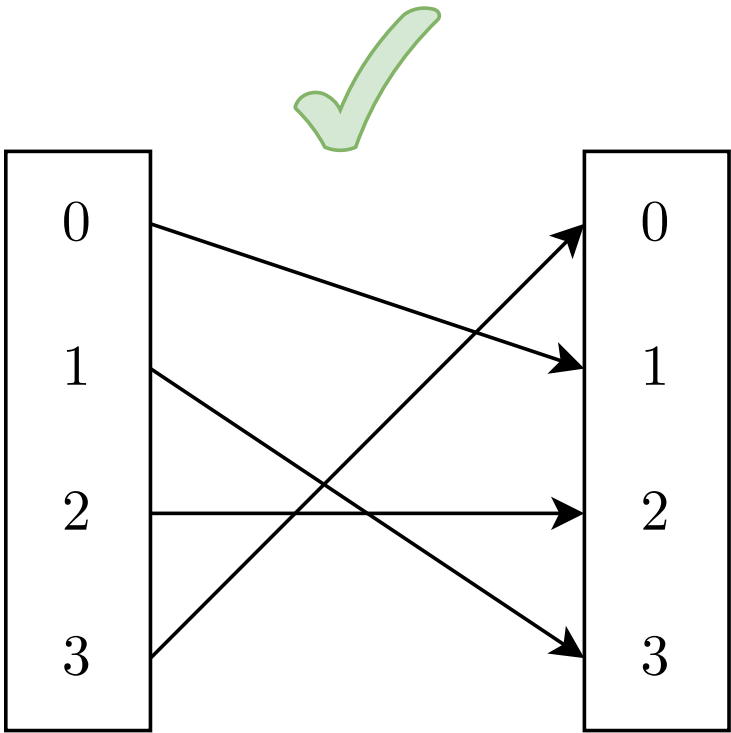
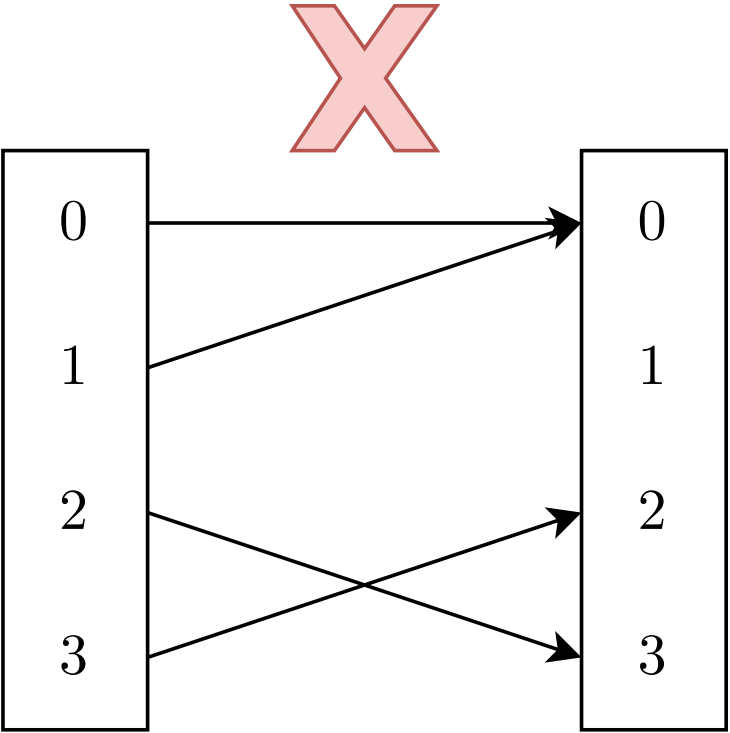
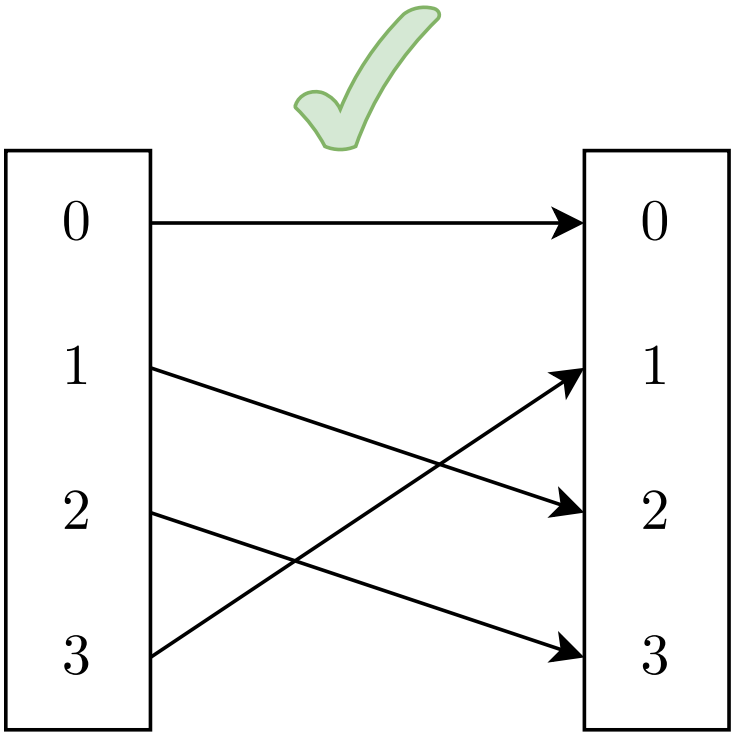
# Binary polynomials

## *Permutation polynomials and invertibility*

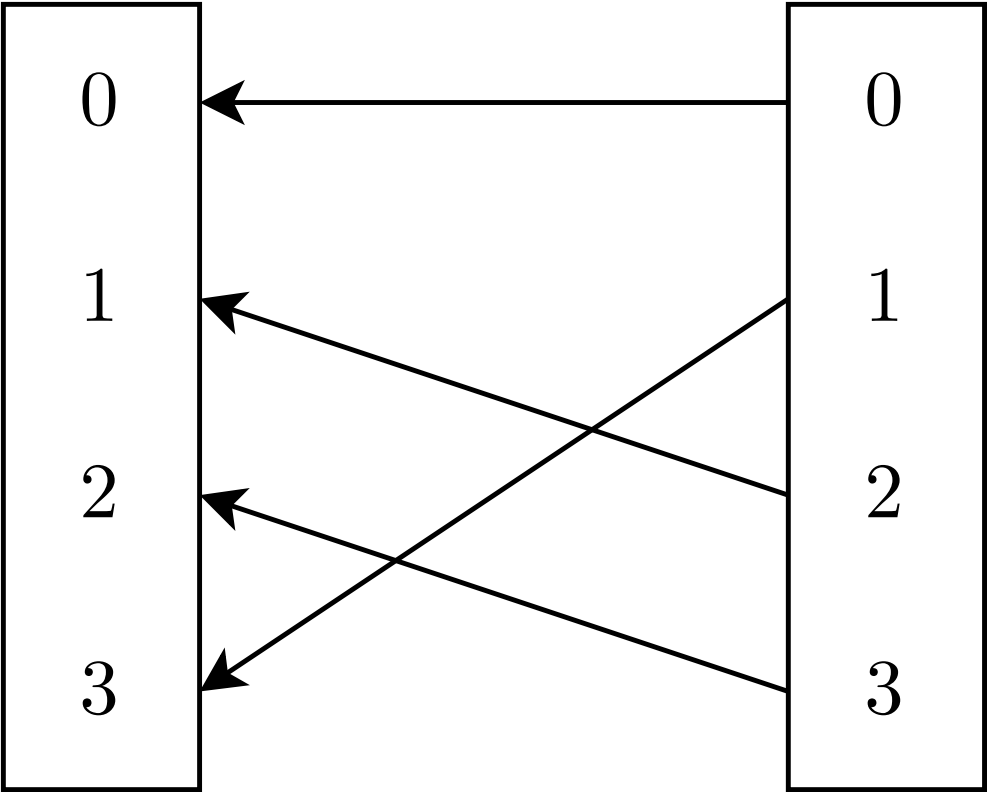
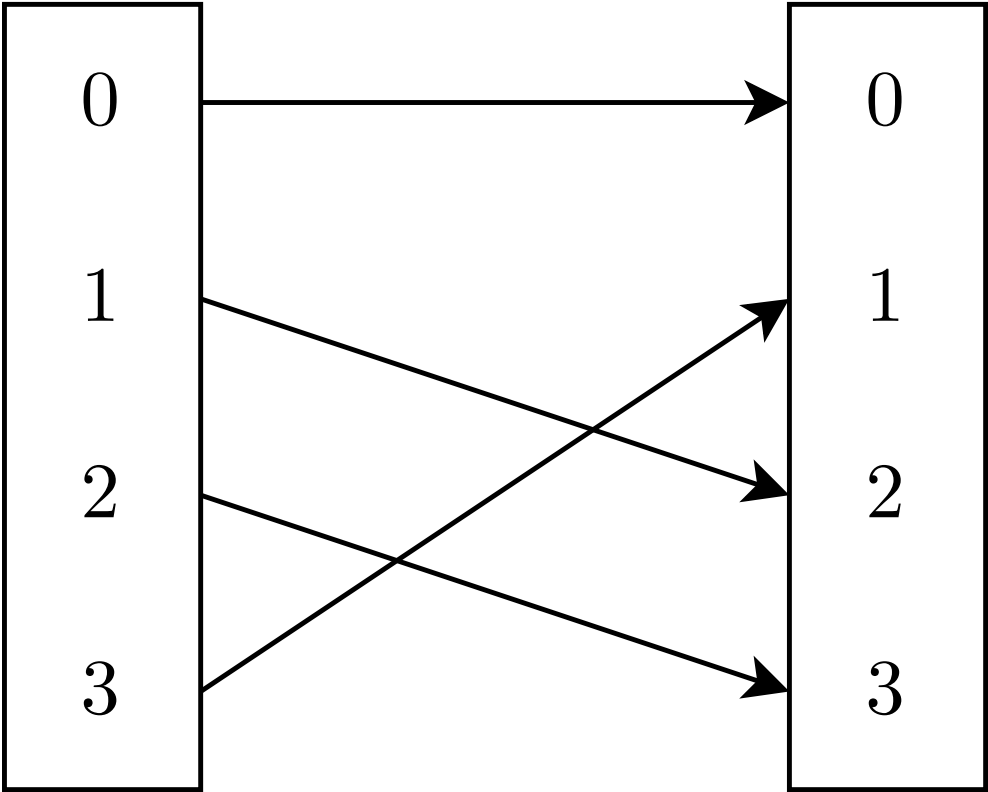
In the univariate case, it makes sense to think about invertibility of binary polynomials.

# Permutation polynomial

A *binary permutation polynomial* is a binary polynomial that defines a bijection: a *one-to-one* mapping; a *permutation*.



By definition, any bijection is invertible; the *inverse permutation*.





The characterization is well-known.

$$P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

$P$  is a binary permutation polynomial modulo  $2^w$ ,  $w \geq 2$  if and only if:

- $a_1$  is odd.
- $(a_2 + a_4 + a_6 + \dots)$  is even.
- $(a_3 + a_5 + a_7 + \dots)$  is even.

# Inversion

Two known public methods to invert general binary permutation polynomials:

- Lagrange-based.
- Newton's inversion algorithm.

Not a fan of either.

I have a method based on Newton interpolation modulo  $2^w$  (not published yet).

## TL;DR

We know how to generate pairs of inverse binary permutation polynomials.

If  $P(x)$ ,  $Q(x)$  are such a pair, then:

$$P(Q(x)) = Q(P(x)) = x$$

for all input values  $x$ .

# Binary polynomials

*Important properties and results*

## Non-unicity of the induced polynomial function

The polynomial function described by evaluating a binary polynomial is **NOT** uniquely determined by its coefficients.

This, a single polynomial function can be defined by multiple polynomial expressions.

Given a binary polynomial

$$P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n \pmod{2^w},$$

it exists a different polynomial (actually infinitely many)

$$Q(x) = b_0 + b_1x + b_2x^2 + \cdots + b_mx^m \pmod{2^w}$$

such that  $P$  and  $Q$  are equivalent as functions ( $P \equiv Q$ ).

In other words,  $P(x) = Q(x)$  for all input values  $x$ .

If we have (infinitely) many binary polynomials that describe the same function, can we choose a single *nice* representative for it?

*Nice* meaning:

- Lowest possible degree.
- Some kind of standardized (normalized) form.

The answer is **YES!**



## Bounded degree of a normalized polynomial representative

We can always find a polynomial of a bounded degree that describes a given polynomial function (with respect to the bitsize  $w$ ).

And we can do so in a standardized procedure. Thus, it makes sense to talk about *reduced* or *normalized* polynomials in a general setting.

If  $w$  is a power of 2 (i.e.,  $w \in \{8, 16, 32, 64, \dots\}$ ), then the max degree required is  $w + 1$ .

In other words, any binary polynomial function in  $\{8, 16, 32, 64, \dots\}$ -bits can be described as a polynomial of degree at most  $\{9, 17, 33, 65, w + 1\}$ .

(Even if the original polynomial that induced the function in place had a way higher degree).

## TL;DR

Given a binary polynomial  $P$ , we know how to:

- Generate functionally equivalent polynomials of arbitrary degree.
- Reduce it to a normalized equivalent representative with guaranteed lowest degree.

## Reference

A. Gàmez-Montolio *et al.*, “Efficient normalized reduction and generation of equivalent multivariate binary polynomials,” in *Workshop on Binary Analysis Research (BAR) at the Network and Distributed System Security (NDSS) Symposium*, Mar. 2024.

 [arnaugamez.com/assets/papers/bar2024.pdf](https://arnaugamez.com/assets/papers/bar2024.pdf)

# Obfuscation

## *Applications of binary polynomials*

Use of equivalent binary (permutation) polynomials as software protection *obfuscation primitives*.

- **Increase diversity:** different versions of the code with equivalent semantics.
- **Watermarking:** different values (coefficients) per build, per client connection, etc.
- **Arbitrary algebraic complexity:** thwart SMT solving, (de)compiler optimizations, etc.

Let's see some ideas and examples.

## Data encodings

Prevent values used in arbitrary computations from being revealed during execution.

1. An initial *encoding* function is first applied to the values to be hidden.
2. An inverse *decoding* function is later applied in combination with (and possibly blended within) the operations that manipulate the initial data.



We can use a pair of inverse permutation polynomials  $P, Q$  as encoding and decoding functions.

```
uint32_t a, b, r;
...
a = key1;
b = key2;
r = foo(a*b);
```

$$P(x) = 1789355803x + 1391591831$$

$$Q(x) = 3537017619x + 624260299$$

$$P(Q(x)) \equiv x$$

$$a = P(key_1)$$

$$b = P(key_2)$$

$$r = foo(Q(a)Q(b))$$

```
uint32_t a, b, r;
...
a = 1789355803*key1 + 1391591831;
b = 1789355803*key2 + 1391591831;
r = foo(4112253801*a*b + 1966380049*a + 1966380049*b + 1062639865);
```

## Insertion of identities

Increase the syntactic complexity of an expression by *wrapping* it with the composition of two inverse functions (i.e., an identity).

We can use a pair of inverse permutation polynomials  $P, Q$  to form such an identity.

```
uint8_t x, y, z;
x = ...;
y = ...;
z = x + y;
```

$$P(x) = 8x^2 + 151x + 111$$

$$Q(x) = 200x^2 + 183x + 223$$

$$P(Q(x)) \equiv x$$

$$z = P(Q(x + y))$$

```
uint8_t x, y, z;
x = ...;
y = ...;
z = 8*(200*(x + y)*(x + y) + 183*(x + y) + 223)*(200*(x + y)*(x + y) +
183*(x + y) + 223) + 151*(200*(x + y)*(x + y) + 183*(x + y) + 223) + 111;
```

## Opaque constants

Conceal (the use of) sensitive constants by replacing them with an expression on an arbitrary number of variables.

The expression will always evaluate to the *opaqued* constant during runtime computation, regardless of the concrete values its variables are assigned to.

We can combine a non-trivially equal to zero MBA expression  $E$  and a pair of inverse permutation polynomials  $P, Q$ .

```
uint8_t k = 123;
foo(k);
```

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

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

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

$$k = P(E(x, y) + Q(k))$$

$$E(x, y) \equiv 0, \quad P(Q(x)) \equiv x$$

```
uint8_t x, y;
x = ...;
y = ...;
foo(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)));
```



## Dynamic (custom) cryptography

In general, you should never roll your own cryptography.

In software protection, it can make (a lot of) sense to roll your own cryptography.

**IT DEPENDS ON THE THREAT MODEL**

Rolling your own cryptography is:

## **A terrible idea**

If your threat model requires strong cryptographic guarantee for the *scrambled data*.

## **An awesome idea**

If your threat model is your adversary discovering the *data scrambling* algorithm you are using (but not if the *scramble* is cryptographically secure).

You do not even have to reinvent the wheel:

1. Pick your favorite standard cryptosystem (e.g., AES).
2. Replace the permutation component(s) (e.g., S-Box) by a (dynamically) generated one defined by a permutation polynomial.
3. Profit.

# Reverse engineering and analysis

## *Applications of binary polynomials*

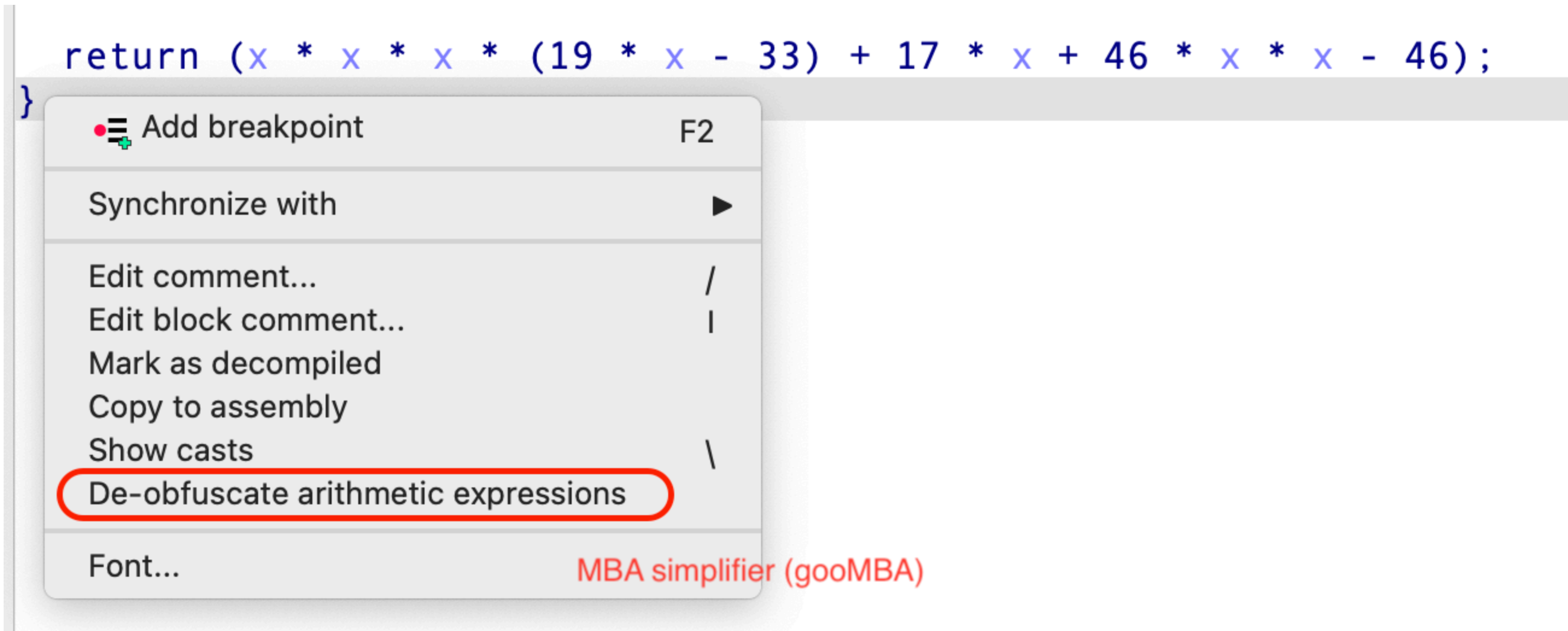
Reduction of (high-degree) polynomial constructs to a normalized and lowest-degree equivalent:

- **Deobfuscation efforts:** post-processing of decompiler output; pre-processing of bit-vector expressions for SMT solvers (usually from symbolic execution).
- **Algebraic manipulation:** standardize expressions in a normal and simple form.
- **Black-box behavior modeling:** represent black-box mappings as (combinations of) normalized binary polynomials.

Let's explore some of that with a demo example.

```
v1 = x * x * x * (19 * x - 33);
v2 = 17 * x + 46 * x * x - 46;
v3 = (v1 | 0xFA) & (v2 | 5);
v4 = v1 ^ v2;
v5 = -48 * (v1 ^ 5 ^ (v1 ^ v2 | 5));
return (-65 * (34 - v3 + v5 * (v4 & 5 ^ v2))
        - 17 * (79 * ((v2 | 0xFA) ^ ~(v1 | 5)) + 16 * (v4 & 5 ^ v1) * (v1 & 0xFA | v2 & 5))
        - 48 * (34 - v3 + v5 * v3) * ((-48 * (v1 & 5 | v2 & 0xFA) - 1) * (v1 & 5 | v2 & 0xFA) + 34)
        + 16
        * (79 * (v1 ^ 0xFA ^ (v4 | 0xFA)) + 16 * (v1 & 0xFA | v2 & 5) * (v4 & 5 ^ v1) + 14)
        * ((16 * ((v1 | 5) & (v2 | 0xFA)) + 79) * (v1 & 0xFA | v2 & 5) - 66)
        + 34);
```





```
v1 = x * x * x;  
v2 = v1 * x;  
v3 = -79 * x + 118 * x * x + 87 * v1;  
v4 = v1 * x * x;  
v5 = v3 + 45 * v2;  
v6 = v4 * x;  
v7 = v5 + 80 * v4;  
v8 = v4 * x * x;  
v9 = v7 - 11 * v6;  
v10 = v8 * x;  
v11 = v9 - 34 * v8;  
v12 = v8 * x * x;  
v13 = v11 - 108 * v10;  
v14 = v12 * x;  
v15 = v13 - 44 * v12;  
v16 = v12 * x * x;  
v17 = v16 * x * x;  
v18 = v15 - 92 * v14 + v16 * x + 27 * v16;  
v19 = v17 * x * x;  
return (v18  
    + 61 * v17  
    - 31 * v17 * x  
    - 60 * v19  
    + 19 * v19 * x  
    + 85 * v19 * x * x  
    + v19 * x * x * x * ((-24 * x - 11) * x + 20)  
    - 46);
```

```
v1 = x * x * x;  
v2 = v1 * x;  
v3 = -79 * x + 118 * x * x + 87 * v1;  
v4 = v1 * x * x;  
v5 = v3 + 45 * v2;  
v6 = v4 * x;  
v7 = v5 + 80 * v4;  
v8 = v4 * x * x;  
v9 = v7 - 11 * v6;  
v10 = v8 * x;
```

```
v11 = v9 - 34 * v8;
```

```
v12 = v8 * x * x;
```

```
v13 = v11 - 108 * v10;
```

```
v14 = v12 * x;
```

```
v15 = v13 - 44 * v12;
```

```
v16 = v12 * x * x;
```

```
return (v15
```



- 92 \* v14

$$+ v_{16} * x$$

+ 27 \* v16

$$+ 61 * v16 * x * x$$
$$- 31 * v16 * x * x * x$$
$$- 60 * v16 * x * x * x * x$$
$$+ 19 * v16 * x * x * x * x * x$$
$$+ 85 * v16 * x * x * x * x * x * x * x$$
$$+ v_{16} * x * x * x * x * x * x * x * x * ((-24 * x - 11) * x + 20)$$

- 46) ;

-  Add breakpoint F2
- Synchronize with 
- Edit comment... /
- Edit block comment... |
- Mark as decompiled
- Copy to assembly
- Show casts \
- De-obfuscate arithmetic expressions**
- Font...

MBA simplifier (gooMBA)

IDA's decompiler + their specific arithmetic simplification engine are really nice for *regular* MBA obfuscation, but accomplish a whole lot of nothing trying to simplify the high-degree polynomial.

Can we do better?

```
from bipo.normalization_uv import NormalizerUnivariate

# Bitsize
w = 8

# Initialize normalizer and inject variable 'x' into scope
NU = NormalizerUnivariate(w)
NU.RR.inject_variables()
```

Defining x

```

# Expression from IDA
v1 = x * x * x;
v2 = v1 * x;
v3 = -79 * x + 118 * x * x + 87 * v1;
v4 = v1 * x * x;
v5 = v3 + 45 * v2;
v6 = v4 * x;
v7 = v5 + 80 * v4;
v8 = v4 * x * x;
v9 = v7 - 11 * v6;
v10 = v8 * x;
v11 = v9 - 34 * v8;
v12 = v8 * x * x;
v13 = v11 - 108 * v10;
v14 = v12 * x;
v15 = v13 - 44 * v12;
v16 = v12 * x * x;
v17 = v16 * x * x;
v18 = v15 - 92 * v14 + v16 * x + 27 * v16;
v19 = v17 * x * x;

P = v18\
    + 61 * v17\
    - 31 * v17 * x\
    - 60 * v19\
    + 19 * v19 * x\
    + 85 * v19 * x * x\
    + v19 * x * x * x * ((-24 * x - 11) * x + 20)\
    - 46

print(f"[+] Recovered polynomial from IDA decompiler:\n---\n{P}")

```

```
[+] Recovered polynomial from IDA decompiler:
```

```
---
```

```

232*x^20 + 245*x^19 + 20*x^18 + 85*x^17 + 19*x^16 + 196*x^15 + 225*x^14
+ 61*x^13 + x^12 + 27*x^11 + 164*x^10 + 212*x^9 + 148*x^8 + 222*x^7 + 24
5*x^6 + 80*x^5 + 45*x^4 + 87*x^3 + 118*x^2 + 177*x + 210

```

This is a binary polynomial of degree 20 over 8-bit values, i.e., modulo  $2^w$ .

But we know that, in the worst case, a polynomial function in 8 bits can be described by a binary polynomial of degree 9.

```
# Normalize the binary polynomial expression  
P_norm = NU.normalize_polynomial(P)  
print(f"[+] Normalized polynomial:\n---\n{P_norm}")
```

```
[+] Normalized polynomial:
```

```
---
```

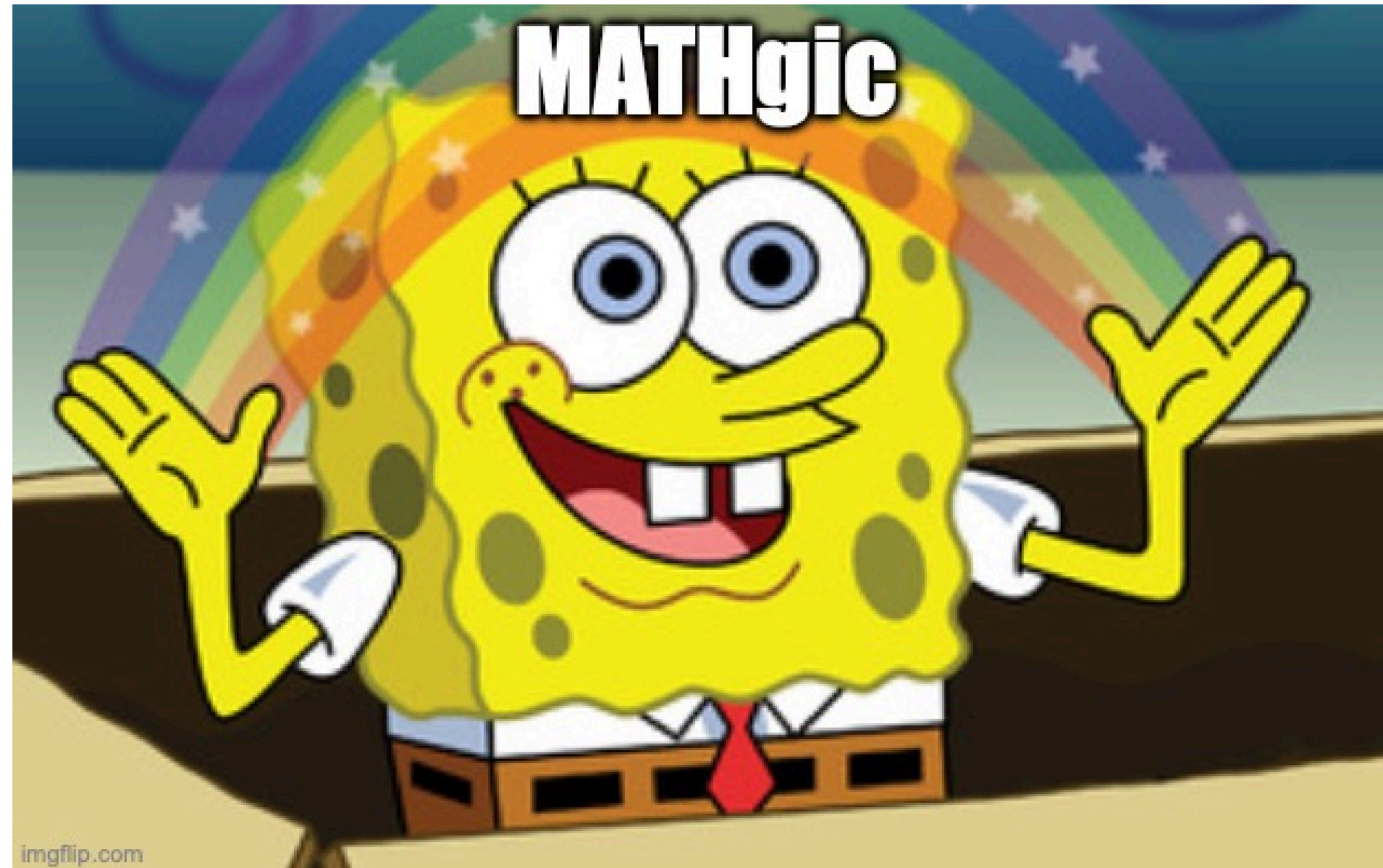
```
19*x^4 + 223*x^3 + 46*x^2 + 17*x + 210
```



```
# Exhaustively assert correctness  
try:  
    for i in range(2w): assert(P(i) == P_norm(i))  
    print("SUCCESS")  
  
except AssertionError:  
    print("FAILURE")
```

SUCCESS

Why this happened?



Arithmetic/MBA simplification engines use a combination of:

- Rewrite rules (pattern matching).
- Basic linear algebra.
- Black-box synthesis of relatively simple expressions.

But the algebraic properties of equivalent binary polynomials are different in nature.

The semantic equivalence is given by class equivalence on the ring on polynomials over the integers modulo  $2^w$  reduced by the ideal of null polynomials modulo  $2^w$ .

(feel free to ignore the previous sentence)

The *bipo* library shown is essentially a bunch of Sagemath wrappers I am building to assist in my exploration of binary polynomials.

Not open sourced (yet?) because of reasons™.

However, my paper is public. And so are the normalization and equivalence generation algorithms described and proved in it.

To this day, there are at least two (public) implementations of the normalization algorithm.





 **ENRGEMBP** Public


 Watch 1



main 1 Branch 0 Tags

+

<> Code

 <b>fvrmatteo</b>	Fixed a bug while printing a Monomial. Made the code PyPy compatible.	121ddcd · 6 months ago	 8 Commits
 README.md	Fixed a bug while printing a Monomial. Made the code P...	6 months ago	
 main.py	Fixed a bug while printing a Monomial. Made the code P...	6 months ago	

 README



 

# Efficient Normalized Reduction and Generation of Equivalent Multivariate Binary Polynomials

This repository contains a pure Python implementation of the `NormalizePolynomial` and `EquivalentPolynomial` algorithms described in the NDSS2024 paper [Efficient Normalized Reduction and Generation of Equivalent Multivariate Binary Polynomials](#) for univariate and multivariate binary polynomials. The

<https://github.com/fvrmatteo/ENRGEMBP>



Mba.Simplifier/Polynomial/PolynomialReducer.cs  +293  ...

```
... @@ -0,0 +1,293 @@
1 + using Mba.Utility;
2 + using System;
3 + using System.Collections.Generic;
4 + using System.Diagnostics;
5 + using System.Linq;
6 + using System.Numerics;
7 + using System.Text;
8 + using System.Threading.Tasks;
9 + using Wintellect.PowerCollections;
10 +
11 + namespace Mba.Testing.PolyTesting
12 + {
13 +     /// <summary>
14 +     /// This class implements the multivariate polynomial reduction algorithm from the NDSS2024 paper
15 +     /// "Efficient Normalized Reduction and Generation of Equivalent Multivariate Binary Polynomials".
16 +     /// </summary>
17 +     public static class PolynomialReducer
18 +     {
```

<https://github.com/mazeworks-security/Simplifier/commit/584956d4b44c3d570de5d49d9d2264c5a24e45fa>

# Conclusions

*Future work*

More binary polynomial algebraic manipulation and speeding up SMT solvers with it.

(Hopefully) more (public) integrations and (efficient) implementations.

Reduction alternatives: sometimes it is faster to build a normalized polynomial from the ground-up than dealing with and reducing an equivalent one of (very) high degree.

- Need fast evaluation and interpolation.
- Working on a couple of papers addressing these topics.

Black-box semantics modeling with binary polynomials:

- This is really cool, but also a vastly unexplored territory.
- Have some early results and conjectures, but nothing ready for public consumption.

# Conclusions

## *Takeaways*

## For software protection folks

- Think of (and build) your transformations in terms of *obfuscation primitives*.
- Do not make protections useless against off-the-shelf tooling: check your binaries!



## For reverse engineers / malware analysts

- Understand what is going on in the software protection field, including academia.
- Try to be one step ahead of the technology available to adversaries (even if not in use yet).

## For people managing teams and budgets

- Slow-cooked research without a short-term ROI is also important, especially in the long run.
- Encourage and facilitate it: give smart people enough time and resources to work on it.

# Thank you

## Q&A

### Archive

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

### Reach out

- [arnaugamez.com](https://arnaugamez.com)
- [linkedin.com/in/arnaugamez](https://linkedin.com/in/arnaugamez)
- [infosec.exchange/@arnaugamez](https://infosec.exchange/@arnaugamez)