Code Deobfuscation

Alban Dutilleul
alban.dutilleul@ens-rennes.fr

Gauvain Thomas
gauvain.thomas@ens-rennes.fr

December 2, 2021

1 Introduction

Code obfuscation is the process of modifying a program, in order to make it unclear and unintelligible (mostly for humans), while remaining functional. It can be achieved for several goals such as hiding the purpose of the software, as well as its behavior and logic, which makes its reverse engineering harder. Either a source code or a binary can be obfuscated. One way to obfuscate one's code is to use a automated dedicated tool, such as Tigress. Tigress is a software, allowing direct obfuscation of C source code in many several ways -of which will be described in this paper- and even allowing combinations and recursive uses of those. Even though these modifications makes it difficult to understand and reverse engineering a code, it is however not impossible to do the obfuscation inverse operation, deobfuscation, which we'll focus on. This project will focus on studying three different ways of Tigress to obfuscate C code, and providing tools to undo them.

2 Analysis and attacks of several Tigress transformations

2.1 Encode Data

2.1.1 General comments

The *EncodeData* transformation replaces integer variables with non-standard representations. Through the `--EncodeDataCodecs` option, it is possible to choose between several representations, although we are only interested in the first one (**poly1**). Indeed the other transformations are not very interesting, they use trivial identities and are therefore very easy to detect.

poly1
: Linear transformation using invertible affine functions $f(x) = ax+b$ (with $a \neq 0$) and $f^{-1}(x) = a^{-1}x - ba^{-1}$ in $\mathbb{Z}/2^n\mathbb{Z}$

xor
: Exclusive-or with a constant using the identity $A \oplus C \oplus C = A$.

add
: Add a constant and promote to next largest integer type. It is not supported for the largest integer type.

If a variable is global an initialization function will be generated and called at the very beginning of the `main`. In the case of the initialization of an array, the first initialized constants are unrolled and the rest will be initialized in a loop to a default value.

```
loop[0] = 1956833575U;
loop[1] = 1956833575U;
loop[2] = 1956833575U;
loop[3] = 1956833575U;
tmp___0 = 4U;
while (! (tmp___0 >= 5U)) {
  loop[tmp___0] = 1206696158U;
  tmp___0 ++;
}
```

All local variables that are encoded in the same scope must have the same type otherwise Tigress raises an error.

2.1.2 In-depth analysis of the poly1 encoding

poly1 computes the modular inverse f^{-1} of an affine function f such that $f^{-1}(f(x)) = x$. Given an integer a and a modulus m, its modular multiplicative inverse is an integer u such that au is congruent to 1 modulo m. A standard way to compute u is to use Euclid's extended algorithm. These operations take place on unsigned integers and are reconverted as needed.

- One way to find the original constant of **poly1** in our code is to find the first place where our variable v is initialized. If the expression has an affine pattern, we can calculate the modular inverse such as $a' \longleftarrow a^{-1} \mod 2^n$ and save $b' \longleftarrow b$. It is possible that an initialization does not contain an affine expression but a constant, in this case we will look further in the code for an affine use as in (2) and we will use it directly in it.

- As soon as we find a use of v within another affine expression, we check that the calculated modular inverse is the right one by comparing it to a', we also check that $(b' \cdot a') \mod 2^n = b$.

- If both conditions are verified, we can then directly replace with the expected value because it was indeed an obfuscated expression. If the expression is nested, we just do a depth-first search by repeating the same process.

2.1.3 Closing remarks

Combining this transformation with *EncodeArithmetic* (which we will present next) makes the problem much more complicated as the expression of the affine function is also modified.

2.2 Encode Arithmetic

2.2.1 Mixed Boolean-Arithmetic expression

This transformation allows to transform a standard arithmetic expression into a Mixed Boolean-Arithmetic (MBA) expression.

Definition 2.1 (MBA expression) An expression E of the form

$$E = \sum_{i \in I} a_i \Big(\prod_{j \in J_i} e_{i,j}(x_1, \dots, x_t)\Big) \qquad (2.2)$$

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 (using the following operators : $\wedge, \neg, \vee, \oplus, \ll, \gg$) of variables $x_1, \dots, x_t$ in $\{0,1\}^n$, $I \subset \mathbb{Z}$ and for all $i \in I$, $J_i \subset Z$ are finite index sets is a Mixed Boolean-Arithmetic (MBA) expression.

Example 2.3 For instance a simple addition is transformed into the following MBA expression by Tigress

$$(x \oplus y) + 2 \cdot (x \wedge y) \qquad (2.4)$$

To generate these expressions, Tigress uses a set of identities that are randomly chosen. They are then applied recursively to a certain depth. It is possible to explore these rules exhaustively by generating a large number of expressions based on the basic operators (see the appendix).
Most of these identities are extracted from [12]. All integer comparison operations are represented by MBA expressions with results in their most significant bit (MSB).

2.2.2 Symbolic simplification of MBAs

We will detail here our approach to simplify the expressions generated by Tigress. The state of the art proposes several types of solutions to simplify generic MBA expressions: pattern matching [8], bit-blasting [9], and program synthesis [3].
The scope of our work being limited to Tigress which has only a limited number of known

rewriting rules led us to choose the pattern matching option. This technique is based on the term rewriting theory that we will briefly describe here.

Definition 2.5 (Term Rewriting System) A Term Rewrite System (TRS) [1] is composed of a set Σ of function symbols and a set $\mathcal{R}$ of rewrite rules over $\mathcal{T}$ (set of terms). A reduction $x \mapsto y$ corresponds to the application of a rule on the term x. A term is said to be normalizing if no other rules can be applied to it.

Definition 2.6 (TRS Properties) A TRS can have several properties, termination, confluence and convergence.

1. **Terminating:** There is no infinite descending chain $a_0 \mapsto a_1 \mapsto \cdots$
2. **Confluent:** When there are different rules to apply from a term t, leading to two different terms y_1 and y_2, we always find a common term z that can be reached from both y_1 and y_2 by successive applications of rewrite rules.
3. **Convergent:** It is both confluent and terminating.

It is therefore relatively straightforward to construct our rewriting system from the Tigress identities, however it is difficult to bring together more than one of the properties stated above. One of the first problems arises from the associativity and commutativity of some arithmetic and bitwise operators ($\{+, \times, \wedge, \vee, \oplus\}$). We can't include these rules in our system unless we create cycles. The solutions proposed in the literature are not easy to implement [1], based for instance on solving homogeneous linear diophantine equations in the non-negatives integers.
If we want to have the convergence property, it implies to determine normal forms, but the complexity of the operations in this TRS makes this difficult to consider. According to Richardson's theorem [11], checking the equivalence of two arithmetic expressions for a certain fairly natural class of expressions is actually an undecidable problem.

It is nevertheless possible thanks to SMT solvers like Z3 [7] to determine in some cases if two expressions are equivalent (for instance: $\neg(a \oplus 23) \equiv (a \oplus 232)$) but it is very costly in performance. This is why we have chosen not to use one.
The rewriting system we propose uses only one structural equality, which increases the number of rules because a new one must be created as soon as an operation is commutative.

2.2.3 The simplification tool: MBA Simplifier

We have implemented our rewriting system in a tool named **MBASimplifier**. It has been realized in the F# functional language, which proposes a powerful pattern matching and a type system facilitating the modeling of a symbolic expression (through sum types).

Project Structure

- **Simplifier.fs** Simplify.fs contains the term rewriting system
- **Ast.fs** The representation of a symbolic MBA expression
- **Engine.fs** The parser of an expression corresponding to a subset of the C99 standard
- **Input.fs**, **UserInterface.fs** Interactions with the user on the standard input
- **Main.fs** Program entry point

The simplification algorithm corresponds to the following pseudocode, a cycle detection algorithm was used to verify that the rule system does not prevent termination.

Classical arithmetic and binary simplification identities have been used in parallel with Tigress specific rules (e.g. $A \vee A \equiv A$).
Since the reduction paths can be long, a hash table of simplifications is used for performance reasons.

Algorithm 1 Eval function pseudocode

```
$expr', \mu, \lambda \longleftarrow$ FLOYDCYCLE($expr$, $simp$)
▹ $\lambda$ is the length of the cycle
if $\lambda > 1$ then EXCEPTION
else
    ▹ Termination condition.
    if $expr' \equiv expr$ then
        return $expr$
    else
        return EVAL($expr'$)
```

Results In order to propose consistent and reproducible results we propose unit tests. Moreover, a script is available to generate an arbitrary number of expressions obfuscated by Tigress, for the moment they have to be compared manually because of the equivalence problem of the expressions mentioned above. However, we propose a measure of the complexity of expressions to quantify the simplification. Expressions are represented as a tree, so we can weight the nodes according to the type of operation.

Type	Weight
Constant	1
Variable	2
Unary operator	4
Binary operator	8

It also is possible to use the dataset named dataset1.c to compare the results on randomly generated expressions, they come from this paper.

Limits of our tool

1. The lack of genericity, if new identities were added, it would be less able to simplify.
2. The implementation of associativity and commutativity which are not optimal.

2.3 Add opaque

This transformation inject superfluous branches (also known as dead branches) in order to complexify the control flow graph (CFG).

2.3.1 Opaque predicates

Collberg et al. introduced the notation of opaque predicates in 1997 [4].

Definition 2.7 (Opaque predicate) An opaque predicate is an invariant expression that always evaluates to true or false with the intention of hiding the fact they are constant.

Collberg differentiates these predicates into several categories according to the difficulty to detect them automatically.

Trivial
A *trivial* opaque predicate is constructed inside a basic block so its invariant expression can be identified at a basic block level.

Weak
A *weak* opaque predicate is constructed throughout a function so it requires intra-procedural analysis to identify its invariant expression.

Strong
A *strong* opaque predicate is constructed across multiple functions so it requires inter-procedural analysis to identify its invariant expression.

Full A *full* opaque predicate is constructed across multiple processes so it requires inter-process analysis to identify its invariant expression.

We will use this classification on the different types of invariants proposed by Tigress.

2.3.2 Invariants generated by Tigress

Tigress allows through the `--AddOpaqueKinds` option to choose among the following invariants:

True
Places a branch that evaluates all the time because the predicate is true.

Junk
Places into a dead branch a sequence of random bytes into the assembly code.

Bug Places a buggified version of its sibling authentic basic block in the unreachable basic block.

Question
Places copy of sibling authentic basic block in the unreachable basic block.

Fake
Places a function call to a non-existing function in the unreachable basic block.

Call Places a function call to a random function existing in executable binary in the unreachable basic.

In addition to that, invariants can use complex predicates via the –AddOpaqueStructs option. Either by using chained lists, or arrays, entropy generated from an external thread but also from the program input.

If we only consider the options on classical data structures, the complexity varies from trivial to weak since the invariants only depend on the function.

Example 2.8 Input-based invariants are not much more complicated to detect as long as we know the inputs. With the following inputs `--Inputs='+2:int:1?2147483647'`, here is the original and generated obfuscated code.

```
str [i] = (( str [ i ] - 'a' +
 ↪  num_shift ) % 26) + 'a';
```

```
atoi_result7 = atoi (*( _ obf_2_ main_
 ↪  _argv + 2) ) ;
if (( atoi_result7 - 1 < 0) + (
 ↪  atoi_result7 - 2147483647 >0) ) {
    *( str + i ) = ( char ) 2;
} else {
    *( str + i ) = ( char ) (((( int )
     ↪  *( str + i ) - 97) + num_shift
     ↪  ) % 26 + 97) ;
}
```

However, thread-based entropy invariants are much more difficult to detect. Tigress uses a Linear congruential generator (LCG) of *Donald Knuth*, the values of the modulus, multiplier and increment are as defined as: $a = 6364136223846793005, c = 1442695040888963407$ [10].

```
case 3:
  __asm__  volatile   ("cpuid\n"
                        "rdtsc\n":
                         ↪  "=a"
                         ↪  (low6),
                         ↪  "=d"
                         ↪  (high7));
  newValue4 = ((unsigned long )high7
   ↪  << 32) | (unsigned long )low6;
  break;
  }
  _1_entropy = (newValue4 + 3) |
   ↪  _1_entropy;
  _1_entropy = 6364136223846793005UL
   ↪  * _1_entropy +
   ↪  1442695040888963407UL;
```

The `_1_entropy` value used in the LCG is the number of cycles that were required to execute the **x86** instruction `cpuid` on the processor.

Using the environment (as with this entropy function) moves an invariant into the strong category.

2.3.3 Proposed Lifting Approach

Since reasoning about invariants is complicated at a high level as in C code, it is reasonable to assume that our analysis will be performed on an intermediate representation to avoid noise related to semantics.

We propose a two-phase approach for Tigress opaque predicates to be removed. Due to its complexity and lack of time, we were unable to implement it.

1. Since invariants can be relatively simple as we have pointed out, it is possible to first statically analyze the code to remove unnecessary branches through the **abstract interpretation** technique (see [5]). This technique has already been used in

the literature for the detection of opaque predicates [6].

2. Since the evaluation of some branches is particularly difficult (typically with the entropy thread), if the static analysis was not sufficient, we can use the **concolic execution** technique to determine which branches are never taken (an example of implementation is proposed in the paper [2]).

3 Conclusion

In our effort, we managed to analyse a small part of the Tigress transformations and to partially lift the expressions generated in **EncodeArithmetic**. However, we realised that it is difficult to completely deobfuscate even a single transformation and even more so when combining them. The literature on the subject is still rather limited and leaves many room for improvement. It is also regrettable that an academic project such as Tigress is not available in open-source, which limits the understanding of some of the techniques it uses.

References

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, USA, 1998.

[2] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded dse: Targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 633–651, 2017.

[3] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 643–659, Vancouver, BC, August 2017. USENIX Association.

[4] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[6] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. Opaque predicates detection by abstract interpretation. In Michael Johnson and Varmo Vene, editors, *Algebraic Methodology and Software Technology*, pages 81–95, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based Obfuscation. In ACM, editor, *2nd International Workshop on Software PROtection*, Vienna, Austria, October 2016.

[9] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions. 2016.

[10] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[11] Dan Richardson and John Fitch. The identity problem for elementary functions and constants. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ISSAC '94, page 285–290, New York, NY, USA, 1994. Association for Computing Machinery.

[12] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

	MBA Expressions	**Identity**
$a \land b$		
1	$(((\lnot a \lor b) + a) + 1)$	$((\lnot a \lor b) - \lnot a)$
2	$((\lnot a \lor b) - \lnot a)$	
$a \lor b$		
1	$(((a + b) + 1) + ((-a - 1) \lor (-b - 1)))$	$((a \land \lnot b) + b)$
2	$((a \land \lnot b) + b)$	
$a \oplus b$		
1	$(((a - b) - ((a \lor \lnot b) + (a \lor \lnot b))) - 2)$	
2	$(((a - b) - ((a \lor \lnot b) \ll 1)) - 2)$	
3	$((a \lor b) - (a \land b))$	
$a \oplus b$		
1	$(((a - b) - ((a \lor \lnot b) + (a \lor \lnot b))) - 2)$	
2	$(((a - b) - ((a \lor \lnot b) \ll 1)) - 2)$	
3	$((a \lor b) - (a \land b))$	
$a + b$		
1	$(((a \oplus \lnot b) + ((a \lor b) + (a \lor b))) + 1)$	
2	$(((a \oplus \lnot b) + ((a \lor b) \ll 1)) + 1)$	
3	$(((a \lor b) + (a \lor b)) - (a \oplus b))$	
4	$(((a \lor b) \ll 1) - (a \oplus b))$	
5	$((a - \lnot b) - 1)$	
6	$((a \oplus b) + ((a \land b) + (a \land b)))$	
7	$((a \oplus b) + ((a \land b) \ll 1))$	
8	$((a \lor b) + (a \land b))$	
$a - b$		
1	$(((a \land \; b) + (a \land \; b)) - (a \oplus b))$	
2	$(((a \land \; b) \ll 1) - (a \oplus b))$	
3	$((a \land \; b) - (\; a \land b))$	
4	$(((a \lor b) \ll 1) - (a \oplus b))$	
5	$((a + \; b) + 1)$	
6	$((a \oplus b) - ((\; a \land b) + (\; a \land b)))$	
7	$((a \oplus b) - ((\; a \land b) \ll 1))$	
$a \times b$		
1	$((a \land b) \times (a \lor b) + (a \land \lnot b) \times (\lnot a \land b))$	
$a = b$		
1	$(\lnot(a - b) \lor (b - a)) \gg b) \land 1$	
$a \neq b$		
1	$((a - b) \lor (b - a)) \gg b) \land 1$	
$a \geq b$		
1	$((\neq (b \oplus a) \gg 1) + (b \land \neq a)) \gg b) \land 1)$	
$a \leq b$		
1	$((\neq (a \oplus b) \gg 1) + (a \land \neq b)) \gg b) \land 1)$	

$a \cdot 2$ 1	$a \ll 1$
$\neg a$ 1	$(-a - 1)$

Table 1: Rewrite rules used by Tigress for generating MBA ex-pressions

It is possible to reproduce these results by going to the `samples/encode_arithmetic` folder and running the bash script `generate.sh` with a large number of iterations ($n \geq 500$). A `out\sorted` folder will be generated with the corresponding identities.