Can a Program Reverse-Engineer Itself?

Antoine Amarilli[1], David Naccache[1], Pablo Rauzy[1], and Emil Simion[2]

[1] École normale supérieure, Département d'informatique
45, rue d'Ulm, F-75230, Paris Cedex 05, France.
{surname.name}@ens.fr
[2] Universitatea din Bucureşti
Faculty of Mathematics and Computer Science
Academiei 14-th, S1, C.P. 010014, Bucharest, Romania
esimion@fmi.unibuc.ro

Abstract. Shape-memory alloys are metal pieces that "remember" their original cold-forged shapes and return to the pre-deformed shape after heating. In this work we construct a software analogous of shape-memory alloys: programs whose code resists obfuscation. We show how to pour arbitrary functions into protective envelops that allow recovering the functions' *exact initial code* after obfuscation. We explicit the theoretical foundations of our method and provide a concrete implementation in Scheme.

1 Introduction

Biological forms of life have a major advantage over machines: the capacity to heal. [4] defines self-healing as *"the property that enables a system to perceive that it is not operating correctly and, without human intervention, make the necessary adjustments to restore itself to normalcy"*. In 3.8 billions of years, natural selection managed to develop outstanding self-healing mechanisms. Living organisms embed biological information in their DNAs. This information and the "defective automaton" represented by a damaged organism manage to heal when damage is not too extreme.

Over the last 50 years considerable efforts were invested in the design of error-correcting codes. Error-correcting codes make it possible to "heal" damaged data with the help of an error-free external decoder. Informally, self-healing systems manage to fix errors even when the decoder's algorithm is somewhat damaged.

In this work we study the design of executable code that resists obfuscation. Our goal is to construct programs that recover their *exact initial code* after obfuscation. In a way, our concept is analogous to shape-memory alloys that "remember" their original cold-forged shapes and return to the pre-deformed shape by heating.

Throughout this paper we will follow the "code is data" notational principle and represent functions $\mathcal{F}$ by the s-expression which is used to define them[3].

2 Real-Life Obfuscation

Informally speaking, an *obfuscator* $\mathcal{O}$ is a function that takes any function $\mathcal{F}$ as an argument and outputs a function $\mathcal{O}(\mathcal{F})$ with an equivalent behavior[4] *i.e.*:

$$\forall x, (\mathcal{O}(\mathcal{F}))(x) = \mathcal{F}(x)$$

Denoting by $\mathbb{F}$ the set of all functions which can be represented by an s-expression, real-life obfuscators usually attempt to transform $\mathcal{F} \in \mathbb{F}$ into an $\mathcal{O}(\mathcal{F}) \in \mathbb{F}$ which is harder to reverse-engineer. Bibliography about the usefulness of obfuscators abounds. We refer the reader to the introductory section of [1] for further reference.

In 2009, Barak *et al.* [1] exhibited a family of unobfuscatable functions $\mathfrak{F}$. Barak *et al.* formalized unobfuscatability by requiring that there exists a property $\pi : \mathfrak{F} \to \{\texttt{true}, \texttt{false}\}$ such that given any program that computes a function $\mathcal{F} \in \mathfrak{F}$, the value $\pi(\mathcal{F})$ can be efficiently computed, while given oracle access to a randomly selected $\mathcal{F} \in \mathfrak{F}$, no efficient algorithm can compute $\pi(\mathcal{F})$ significantly better than random guessing.

We start by observing that given any "real-life" (*i.e.* commercial) obfuscator, the construction of inherently unobfuscatable code is easy: A Quine (named after the logician Willard Van Orman Quine) is an unobfuscatable program[5] that prints its own code [2,5].

Writing Quines is a somewhat tricky programming exercise yielding Lisp, C or natural language examples such as:

```
((lambda (x) (list x (list (quote quote) x)))
(quote (lambda (x) (list x (list (quote quote) x)))))
```

```
char *f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";
main() {printf(f,34,f,34,10);}
```

```
Copy the next sentence twice. Copy the next sentence twice.
```

A Quine $\mathcal{Q}$ is impossible to obfuscate because either the evaluation of $\mathcal{O}(\mathcal{Q})$ yields $\mathcal{Q}$ and hence reveals the original pre-obfuscation code (thereby making

[3] *i.e.* Whenever we write that a function is taken as argument or returned by another function, we really mean that s-expressions are taken and returned. Note that instead of s-expressions, one could use λ-terms or any form of source code.

[4] We write $\mathcal{F}(x) = \perp$ if $\mathcal{F}(x)$ does not terminate.

[5] Any Turing-complete formal system admits Quines [3].

obfuscation impossible) or it does not – in which case $\mathcal{O}$ is not a valid obfuscator because then we would have that

$$(\mathcal{O}(\mathcal{Q}))(x) \neq \mathcal{Q}(x)$$

This gives hope to construct in a somewhat generic manner unobfuscatable versions of arbitrary functions. Namely, if we could design a "Genetically Modified Organism" hybridizing a Quine $\mathcal{Q}$ and an arbitrary function $\mathcal{F}$, then one may reasonably hope that the resulting version of $\mathcal{F}$ will inherit the obfuscation-resistance features of the Quine while still performing the calculations that $\mathcal{F}$ encodes. This is the question dealt with by the present paper.

3 The Construction

Consider the function $\mathcal{W}$ such that for all functions $\mathcal{F} \in \mathbb{F}$ and all inputs x :

$$(\mathcal{W}(\mathcal{F}))(u, x) = \begin{cases} \mathcal{F} & \text{if } u = \texttt{true} \\ \\ \mathcal{F}(x) & \text{if } u = \texttt{false} \end{cases}$$

Given our definition of obfuscation, $\mathcal{W}(f)$ cannot be obfuscated in any meaningful way, simply because all obfuscators $\mathcal{O}$, must still ensure that

$$(\mathcal{O}(\mathcal{W}(\mathcal{F})))(\texttt{true}, x) = \mathcal{F}$$

Let's see what happens if we relax the setting and allow $\mathcal{O}$ to obfuscate the result of $W(\mathcal{F})(\texttt{true}, x)$. In this case, $\mathcal{O}$ could proceed by retrieving $\mathcal{F} = (\mathcal{W}(\mathcal{F}))(\texttt{true}, x)$, build an $\mathcal{O}(\mathcal{F})$ such that $(\mathcal{O}(\mathcal{F}))(x) = \mathcal{F}(x)$ for all x, and then build an $\mathcal{O}(\mathcal{W}(\mathcal{F}))$ such that:

$$(\mathcal{O}(\mathcal{W}(\mathcal{F})))\,(u, x) = \begin{cases} \mathcal{O}(\mathcal{F}) & \text{if } u = \texttt{true} \\ \\ \mathcal{F}(x) & \text{if } u = \texttt{false} \end{cases}$$

to prevent us from recovering the original $\mathcal{F}$. However, we would still like $\mathcal{O}$ to return a function which is *equivalent* to the original $\mathcal{F}$.

Let us define formally what we mean by *equivalent* in the previous sentence: We write

$$\mathcal{F} =_0 \mathcal{F}'$$

if $\mathcal{F}$ and $\mathcal{F}'$ are the exact same s-expressions (*i.e.* the same executable code).

We write $\mathcal{F} =_1 \mathcal{F}'$ if $\mathcal{F}$ and $\mathcal{F}'$ have an equivalent *behavior i.e.*

$$\forall x, \mathcal{F}(x) = \mathcal{F}'(x)$$

For all $n > 1$, we define

$$\mathcal{F} =_n \mathcal{F}' \Leftrightarrow \forall x, \mathcal{F}(x) =_{n-1} \mathcal{F}'(x)$$

with $\bot =_n x$ iff $x =_0 \bot$.

We extend the notation to arbitrary values (not necessarily programs):

$$x =_n y \Leftrightarrow x = y$$

and to tuples:

$$(x_i)_i =_n (y_i)_i \Leftrightarrow x_i =_n y_i \quad \text{for all } i$$

Now, instead of requiring that $\mathcal{O}(\mathcal{F}) =_1 \mathcal{F}$ for all x, we fix some constant n and require that $\mathcal{O}(\mathcal{F}) =_n \mathcal{F}$.

Is it possible to build unobfuscatable programs under the above definition?

While intricate, the answer turns out to be positive, for any n.

To do so, we define the function $\mathcal{C}$ such that $\forall \mathcal{F} \in \mathbb{F}$ and for all inputs x:

$$(\mathcal{C}(\mathcal{F}))(u, x) = \begin{cases} \mathcal{C}(\mathcal{F}) & \text{if } u = \texttt{true} \\ \mathcal{F}(x) & \text{if } u = \texttt{false} \end{cases}$$

Note that this requires $\mathcal{C}(\mathcal{F})$ to reference its own source code, which can be done by a Quine-like construction similar to the one used in Pastis [6] and which is justified theoretically by Kleene's second recursion theorem.

We claim that $\mathcal{C}(\mathcal{F})$ cannot be obfuscated in a way respecting the above constraints, no matter one's choice of n.

Indeed, given an obfuscated version $\mathcal{O}(\mathcal{C}(\mathcal{F})) =_n \mathcal{C}(\mathcal{F})$, we can invoke

$$(\mathcal{O}(\mathcal{C}(\mathcal{F})))(\texttt{true}, x)$$

and obtain a function which is $=_{n-1} \mathcal{C}(\mathcal{F})$. We execute this "peeling process" n times and end up with a function which is $=_0 \mathcal{C}(\mathcal{F})$, *i.e.* the original $\mathcal{C}(\mathcal{F})$, from which $\mathcal{F}$ can be retrieved. We denote by $\mathcal{D}_n$ a function implementing this "peeling process".

In summary $\forall n$, there exists two functions $\mathcal{C}$ and $\mathcal{D}_n$ such that $\forall \mathcal{F} \in \mathbb{F}$, for any obfuscator $\mathcal{O}$ verifying $\mathcal{O}(\mathcal{F}') =_n \mathcal{F}'$ for all $\mathcal{F}'$, we have:

$$(\mathcal{O}(\mathcal{C}(\mathcal{F})))(\texttt{false}, x) =_n \mathcal{F}(x) \quad \text{and} \quad \mathcal{D}_n(\mathcal{O}(\mathcal{C}(\mathcal{F}))) = \mathcal{F}$$

4 oximoron: Clear Obscure Code Implementation

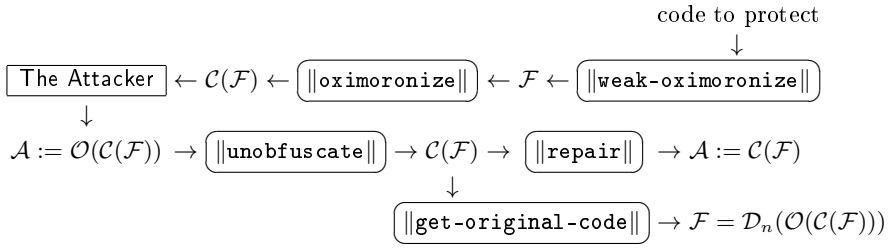
`oximoron` is written in Scheme[6], a Lisp variant. The code, given in Appendix A[7], defines a macro `weak-oximoronize` implementing $\mathcal{F}$ and a macro `oximoronize` implementing $\mathcal{C}$.

The macro `call` calls a (possibly obfuscated) $\mathcal{O}(\mathcal{C}(\mathcal{F}))$ with arguments $(\texttt{false}, x)$ to get a (possibly obfuscated) $\mathcal{O}(\mathcal{F}) \leftarrow \mathcal{O}(\mathcal{C}(\mathcal{F}))(\texttt{false}, x)$.

The function $\mathcal{D}_n$ is implemented in three steps:

- `unobfuscate` retrieves $\mathcal{C}(\mathcal{F})$ from a (possibly obfuscated) $\mathcal{O}(\mathcal{C}(\mathcal{F}))$.
- `repair` alters the environment to replace a (possibly obfuscated) $\mathcal{O}(\mathcal{C}(\mathcal{F}))$ by the original $\mathcal{C}(\mathcal{F})$.
- Finally, `get-original-code` extracts $\mathcal{F}$ from $\mathcal{C}(\mathcal{F})$.

In other words, the defender and the attacker (obfuscator) perform the following sequence of operations:



5 Further Research

A very interesting challenge would be to design nontrivial functionality-preserving[8] programs and obfuscators. Letting π be a nontrivial property $\pi : \mathcal{F} \rightarrow \{\texttt{true}, \texttt{false}\}$, we define $\mathcal{O}_\pi$ as a functionality-preserving obfuscator with respect to property π if $\pi(\mathcal{O}_\pi(\mathcal{F})) = \pi(\mathcal{F})$.

For instance, a functionality-preserving Quine would be a code $\mathcal{Q}$ such that $\mathcal{O}_\pi(\mathcal{Q})$ prints $\mathcal{O}_\pi(\mathcal{Q})$. In this example, the property $\pi(\mathcal{F})$ is the answer to the question "Is $\mathcal{F}$ a Quine?" (instead of the question "Is $\mathcal{F}$ a *specific* Quine?").

Functionality-preserving obfuscation generalizes classical (function-preserving) obfuscation because in a classical obfuscator, π attempts to answer a question relating to the mathematical function F (encoded in the program $\mathcal{F}$) and not to the actual code of $\mathcal{F}$ (that computes F).

---

[6] We use the Racket platform (`racket-lang.org`), a descendant from Scheme and a programming language research platform.

[7] The code can also be downloaded from `http://pablo.rauzy.name/files/oximoron.zip`

[8] Rather than *function-preserving*.

References

1. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan and K. Yang, On the (Im)possibility of Obfuscating Programs, Advances in Cryptology - CRYPTO'01, vol. 2139 of Lecture Notes in Computer Science, pp. 1–18, 2001. Springer-Verlag
2. J. Burger, D. Brill and F. Machi, Self-reproducing programs, Byte, volume 5, August 1980, pp. 74—75.
3. N. Cutland, Computability: An introduction to recursive function theory, Cambridge University Press, pp. 202–204, 1980.
4. D. Ghosh, R. Sharman, H. Rao, S. Upadhyaya, Self-healing systems — survey and synthesis, Decision Support Systems 42 (2007) pp. 2164-–2185
5. D. Hofstadter, Godel, Escher, and Bach: An eternal golden braid, Basic Books, Inc. New York, pp. 498—504.
6. A. Amarilli, S. Müller, D. Naccache, D. Page, P. Rauzy and M. Tunstall, *Can Code Polymorphism Limit Information Leakage?*, Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication - 5th IFIP WG 11.2 International Workshop, WISTP'11, vol. 6633 of Lecture Notes in Computer Science, pp. 1–21, 2011. Springer-Verlag.

A The Source Code

```
(module oximoron racket/base

  ;; The name oximoron come from "oxymoron", because we can have clear
  ;; obscure code (clarifiable obfuscated code).

  (define-syntax-rule (weak-oximoronize function)
    #; "weak oximoron"
    '(lambda (unobfuscate? . args)
       (if unobfuscate?
           'function
           (apply function args))))

  (define-syntax-rule (oximoronize function)
    #; "strong oximoronize using quine+payload (like PASTIS but next-gen)"
    '(lambda (unobfuscate? . args)
       (define (Q expr)
         `(lambda (unobfuscate? . args)
            ,@expr
            (if unobfuscate?
                (Q '(,@expr))
                (apply function args))))
       (if unobfuscate?
           (Q '((define (Q expr)
                  `(lambda (unobfuscate? . args)
                     ,@expr
                     (if unobfuscate?
                         (Q '(,@expr))
                         (apply function args))))))
           (apply function args))))

  (define-syntax-rule (call oximoron args ...)
    #; "simulate classic function call"
    (oximoron #f args ...))

  (define-syntax repair
    #; "repair the oximoron"
    (syntax-rules ()
      ([_ oximoron]
       (set! oximoron (eval (oximoron #t))))
      ([_ oximoron level]
       (let loop ([n level])
         (cond
          ([> n 0]
           (repair oximoron)
           (loop (sub1 n))))))))

  (define-syntax unobfuscate
    #; "return the oximoron code"
    (syntax-rules ()
      ([_ oximoron]
       (oximoron #t))
      ([_ oximoron level]
       (let loop ([n level] [oxi oximoron])
         (cond
          ([zero? n] (unobfuscate oxi))
          ([> n 0]
           (loop (sub1 n) (eval (unobfuscate oxi)))))))))

  (define (get-original-code oximoron [level 0])
    #; "get the original function code (before oximoronization)"
    (cadr (cadddr (cadddr (unobfuscate oximoron level)))))

  )
```

B A REPL Session

Here is a REPL[9] session demonstrating `oximoron`'s usage. The REPL session shows that the obfuscated version of the oximoronized `fib` contains an obvious copy of the unobfuscated `fib` which at a first glance may look as cheating! A careful look into the code reveals that the copy of the unobfuscated code is a quoted *literal data value* injected into the oximoronized `fib` to allow later recovery. If the obfuscator alters these literal data values, this will alter the computed function rather than the way in which it is computed (and will hence contradict the way in which obfuscation is defined – *i.e.* this would result in a definition-incompliant obfuscator *modifying* rather than *re-writing* its input).

By a way of analogy, in the movie RoboCop, the cyborg RoboCop obeys by $4 = 3+1$ directives: serve the public trust, protect the innocent, uphold the law and a classified fourth directive that prevents RoboCop from arresting or harming any senior executive of the mega-corporation OCP. As the movie ends, the bad character, who is an OCP executive, grabs a gun and takes the president hostage. While RoboCop recognizes a violation of directives 2 and 3 he cannot intervene by virtue of directive 4. Here as well, whilst the obfuscator may recognize the original code in the literal data value it cannot intervene precisely because... it is a obfuscator!

```
> (oximoronize (lambda (arg)
    (let fib ([n arg])
      (if (< n 2)
          n
          (+ (fib (- n 1)) (fib (- n 2)))))))

'(lambda (unobfuscate? . args)
   (define (Q expr)
     `(lambda (unobfuscate? . args)
        ,@expr
        (if unobfuscate?
          (Q '(,@expr))
          (apply
           (lambda (arg)
             (let fib ((n arg))
               (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
           args))))
   (if unobfuscate?
     (Q
      '((define (Q expr)
          `(lambda (unobfuscate? . args)
             ,@expr
             (if unobfuscate?
               (Q '(,@expr))
               (apply
                (lambda (arg)
                  (let fib ((n arg))
                    (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
                args))))))
     (apply
      (lambda (arg)
        (let fib ((n arg)) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
      args)))

> ; let's define an obfuscated version of fib:
> (define fib (lambda (__ . _)
    (define (Q --)
```

[9] Read-Eval-Print Loop

```
      `(lambda (_ . __)
         ,@--
         (if _
           (Q '(,@--))
           (apply
            (lambda (_____)
              (sleep 3)
              (let _-_ ((_ _____))
                (if (< _ 2) _ (+ (_-_ (- _ 1)) (_-_ (- _ 2))))))
            __))))
    (if __
      (Q
       '((define (Q expr)
           `(lambda (unobfuscate? . args)
              ,@expr
              (if unobfuscate?
                (Q '(,@expr))
                (apply
                 (lambda (arg)
                   (let fib ((n arg))
                     (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
                 args))))))
      (apply
       (lambda (_)
         (sleep 5)
         (let -_ ((__- _))
           (if (< __- 2) __- (+ (-_ (- __- 1)) (-_ (- __- 2))))))
       _))))
> (time (call fib 10))
cpu time: 4 real time: 5002 gc time: 0
55
> (unobfuscate fib) ; one level unobfuscation
'(lambda (_ . __)
   (define (Q expr)
     `(lambda (unobfuscate? . args)
        ,@expr
        (if unobfuscate?
          (Q '(,@expr))
          (apply
           (lambda (arg)
             (let fib ((n arg))
               (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
           args))))
   (if _
     (Q
      '((define (Q expr)
          `(lambda (unobfuscate? . args)
             ,@expr
             (if unobfuscate?
               (Q '(,@expr))
               (apply
                (lambda (arg)
                  (let fib ((n arg))
                    (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
                args))))))
      (apply
       (lambda (_____)
         (sleep 3)
         (let _-_ ((_ _____)) (if (< _ 2) _ (+ (_-_ (- _ 1)) (_-_ (- _ 2))))))
       __)))
> (unobfuscate fib 2) ; two level unobfuscation
'(lambda (unobfuscate? . args)
   (define (Q expr)
     `(lambda (unobfuscate? . args)
        ,@expr
        (if unobfuscate?
          (Q '(,@expr))
          (apply
```

```
          (lambda (arg)
            (let fib ((n arg))
              (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
          args))))
   (if unobfuscate?
     (Q
      '((define (Q expr)
          `(lambda (unobfuscate? . args)
             ,@expr
             (if unobfuscate?
               (Q '(,@expr))
               (apply
                (lambda (arg)
                  (let fib ((n arg))
                    (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
                args))))))
     (apply
      (lambda (arg)
        (let fib ((n arg)) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
      args)))
> ; we recognize our oximoronized version of fib.
> (get-original-code fib 2)
'(lambda (arg)
   (let fib ((n arg)) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
> (repair fib 2) ; two level reparation
> (time (call fib 10))
cpu time: 0 real time: 1 gc time: 0
55
> ; if we would have called (repair fib) instead of (repair fib 2)
> ; (call fib 10) ``real time'' would have taken 3 more seconds.
```