On Entropy Measures for Code Obfuscation

Roberto Giacobazzi and Andrea Toppan
University of Verona
E-mail: (roberto.giacobazzi | andrea.toppan)@univr.it

***Abstract*—The purpose of this paper is to introduce a further measurement for software obfuscation, in particular observing that many important obfuscation transformations increase the uncertainty an attacker has about the program behaviour, uncertainty modeled by the entropy of the program traces or the nodes under execution. The transformations considered in this paper are unknown opaque predicates insertions or unknown dispatcher insertions, where the latter are an extension of the if-else statements of unknown opaque predicates to switch-case statements. Consequences of modeling obfuscation as an increase of entropy can be simple guidelines to obtain potent transformations at low cost and the explanation of existing transformations effectiveness. We present a program transformation algorithm based on the latter observations.**

***Index Terms*—Code obfuscation, software protection, and information theory**

I. Introduction

Software complexity metrics, well fitting in the definition described as transformation potency [18], are also important ways to describe and measure complexity from a design point of view, to determine software similarity and to adopt testing methodologies effective in reducing bug count.

The purpose of software obfuscation (see [7] for a comprehensive survey) is the protection of information contained in a program (e.g., for intellectual property) and control-flow obfuscation explicitly aims at increasing the Control Flow Graph (CFG) intricacy in the most effective way to confuse attackers. Although perfect and universal obfuscation is impossible [3], code obfuscation still represents a key technological asset in modern protection, e.g., for hiding keys and sensitive informations in software. The importance of having accurate quantitative metrics for estimating the potency of code obfuscation against Man-At-The-End (MATE) attacks has been addressed in [6]. Methods and algorithms have been introduced for obscuring code most deriving from an intuitive, in some cases ad-hoc, models of software interpretation (e.g., in a MATE scenario) and justified by employing standard software engineering metrics for estimating the degree of potency of the obfuscation [11], such as: the program length, its cyclomatic complexity (the number of predicates in a program), the nesting, data-flow, data-structure and inheritance complexity, etc. An example of metric combining data-flow and program length is found in [25], where the most relevant contribution to obfuscation is shown to be a virtual mental simulation method doing variable backtracking, particularly expensive for non-constant variables and off course small queue sizes resembling human short-term memory capabilities. Measures of program obfuscation can relate to instruction count, data-flow, control-flow as distinguished in [11], where a good emphasis is put on the so called *cyclomatic complexity*, *knot count* and *instruction count*. According to the cyclomatic complexity [19], an unknown opaque predicate (see [8]) generates an increase of complexity of 1. Figure 1 gives an example with a simple graphical description about it: The sequential piece of code to be obfuscated is substituted by an introductory piece also computing a value tested for a condition, connected to the two alternative sequential execution paths, connecting to an un-obfuscated piece of code that completes the functionality of the original piece, connecting to the subsequent node of the original program. The overall increase of the cyclomatic complexity is given by

$$[(e+4)-(n+3)+2\cdot c]-[e-n+2\cdot c]=1$$

where e is the number of edges, n is the number of nodes and c is the number of connected components. Generalizing it to a k-way dispatcher, the increase is given by

$$[(e+2\cdot k)-(n+1+k)+2\cdot c]-[e-n+2\cdot c]=k-1$$

where k is the number of alternative branches. More recent developments have shown that it is possible to systematically derive efficient obfuscating algorithms by distorting interpreters [13] yet providing a semantics-based qualitative analysis of their effectiveness in terms of the expected effort that an approximate (automatic) attacker (interpreter) has to make in order to disclose the intended behavior of the transformed code [12]. This however does not provide any quantitative measure of the potency of an obfuscating transformation, and still the finding of adequate metrics that drive the construction of obfuscating algorithms is a challenge.

In this paper we go beyond standard complexity measures by considering the increasing of the entropy as the key measure associated with code obfuscation. The theoretical support of this paper consists in probabilistic programs, intended both as programs that process input distributions to produce output distributions and as programs that randomly change their semantics even on fixed inputs [16]. The entropy considered in this paper deals with combinatorial aspects of a program rather than communication. We introduce two entropy measures for CFG obfuscating transformations, in particular for opaque predicate insertion [8] and code flattening via dispatcher insertion [14]. We show that relevant criteria for increasing software obscurity are justified and can be systematically improved from the analysis of these metrics. They can be used to intentionally produce programs having highly unpredictable behaviors (i.e., sequences of instructions or traces of control-flow points), although we believe they represent a point of view suitable to become a software metrics

by itself. We introduce this idea through an example.

A. Program example and explanation of the new approach

Consider the program computing the distance from the origin of a point in the Cartesian plane (javascript example).

```
function D() {
 x=document.A.B.value;
 y=document.A.B2.value;
 n=0;
 n=Math.sqrt(x*x+y*y);
 document.write(n+"<br/>");
return n;
}
```

After an obfuscation transformation (whose details we omit) we may obtain the following code. The first thing that the transformation evidences is not only a longer program, but also many different sequential portions of code (about three new ones), basically a program that is not as easy to memorize and grasp as the program before the transformation.

```
function d() {
x=document.A.B.value;
y=document.A.B2.value;
b=new Boolean();
i=0;
j=0;
k=0;
n=0;
nn=0;
          while(i<x) {
          k=k+i;
          i=i+1;
          j=j+i;
          }
          nn=j+k;
          i=0;
          k=0;
          j=0;
          b=nn==16;
          if(b) {
          nn=nn+y*y;
          }
          else {
                    while(i<y) {
                    k=k+i;
                    i=i+1;
                    j=j+i;
                              }
          nn=nn+j+k;
          }
n=Math.sqrt(nn);
document.write(n+"<br/>");
return n;
}
```

In this paper we model programs both as sources of different nodes and as source of traces (intended as sequences of control-flow outcomes). The source corresponding to the first program, e.g., analyzed in a debugger, transmits a sequence of symbols (command lines or code blocks) which is increased in the second program due to obfuscation. Moreover, a conditional statement in the transformed program shows that the program traces are more than the original one: *a higher unpredictability in overall program behavior can be pointed out by measuring its entropy*. Random behavior corresponds here to the setting of a random watch point along the execution of the two programs, as typically done in a debugging activity for reverse engineering, e.g., by stopping program execution and watching memory contents. Another entropy measure can be associated with the control flow execution. Here the randomness is associated with the probability of a sequence of transitions of control points generated from a source of inputs with a given (e.g, equivalent) probability. In the first program there is a sequential piece of code that computes a distance after variables initializations (the sequential piece of code has probability one and entropy zero). In the second program there are six different sequential pieces of code having probability greater than zero (entropy greater than zero). In the first program there is only one trace: it has zero entropy no matter the inputs. Changing the inputs, the second program has two reachable traces (one associated with the if path, the other with the else path): the entropy is greater than zero.

B. Preliminaries

For an exhaustive introduction to information theory and random variables see [15]. The concept used in this paper is the one of a discrete random variable. Given a finite and discrete set of elements $\mathcal{X} = \{x_1, x_2, ..., x_k\}$, named alphabet, a random variable X is characterized by its distribution

$$P_X = \left\{ \; p_X(x) \in [0,1] \,\middle|\, \; x \in \mathcal{X} \; \right\}$$

where $p_X(x) = P[X = x] = p_x$, that is the probability of X taking the value x belonging to the alphabet $\mathcal{X}$. Each p_x is nonnegative and $\sum_{x\in\mathcal{X}} p_x = 1$. Shannon's Entropy [22] is defined as

$$H(X) = -\sum_{x\in\mathcal{X}} p_x \cdot log_2(p_x)$$

Moreover, conditional probabilities are defined as

$$P[A|B] = \frac{P[AB]}{P[B]}$$

where $P[A|B]$ is the probability of event A conditioned by event B, and $P[AB]$ is the probability of events A and B. When A and B are independent $P[AB] = P[A] \cdot P[B]$ and P[A|B]=P[A]. The probability of union of disjoint events $A \cap B = \emptyset$ is given by $P[A \cup B] = P[A] + P[B]$, which is the probability that either A or B happens.

II. Modeling programs as information theoretic white-box sources of control-free blocks

Among the existing models for the analysis of programs the most complete ones are found in [20] (e.g. discrete dynamic systems). The field of this work tackles reverse engineering, which can become a very hard game played by dynamic attackers of a program that take several glimpses on it by mean of advanced automatic tools, and pause at most nodes to figure them out: it is the situation in which a solution like [2] is supposed to put an attacker in: *forcing* a limited local visibility of the control-flow graph. We begin introducing a simple notion of entropy, that takes into account only local computations, defining a control-free block of code, that is

different from basic node because it refers essentially to non-branch instructions. However, during this work, they will be used interchangeably. A *c*ontrol-free block is a piece of program beginning with the first line of its code or preceded by a label; ending with branch instruction to a target label or a jump; containing no other branch or jump instruction, labels or syscalls in between. At a first glance, the most simple and ingenuous information theoretic model of a program P is a source of control-free blocks of code, a mere collection of nodes with unknown connecting edges:

$$\mathcal{A}_P = \left\{n_1, n_2, \ldots, n_{ob}, \ldots, n_{|A_P|}\right\}$$

where the n_i are the nodes of the CFG of P and n_{ob} is the node undergoing a branch insertion transformation T (i.e., unknown opaque predicate insertion) like in Figure 1. Affected nodes are double-circled nodes, the others being unaffected.

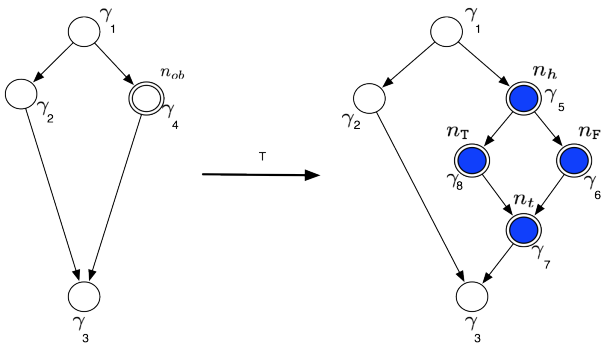

Figure 1. Unknown binary opaque predicate insertion

Therefore, after a transformation T we get for the transformed program $P' = T(P)$ another source featuring an alphabet having bigger cardinality, containing *new elements* n_{ob_T},n_{ob_F}, n_h and n_t instead of n_{ob}:

$$\mathcal{A}_{P'} = \left\{n_1, n_2, \ldots, n_{ob_F}, n_{ob_T}, n_h, n_t, \ldots, n_{|A'_P|}\right\}$$

where n_h is the top new node, n_t the bottom new node, $n_{ob_T}$the node of the true path, $n_{ob_F}$the node of the false path.

A white-box attacker of program P observes the outcomes of a discrete random variable ranging in $\mathcal{A}_P$ named *control-free block of program P:* $B(P)$. For each of these variables, consider the distribution:

$$P_{B(P)} = \left\{\ p_{B(P)}(b) \in [0,1] \,\middle|\, b \in \mathcal{A}_P\ \right\}$$

where $p_{B(P)}(b)$ is the probability that the random variable $B(P)$ has value b, namely it is the probability that the execution of a program instruction (apart from branches and jumps) belongs to the corresponding control-free block in the program. The following theorem specifies that code insertion increases the entropy of the $B(P)$ source.

Theorem 1: Branch insertion transformations, introducing new nodes in the CFG of a program P, increase the entropy of $B(P)$.

Proof. Denote by P' the transformed program obtained by (opaque) branch insertion. Observing a new branch-reachable node n_{ob_i} instead of the complete node n_{ob} in the original program P is an event having the following probability [1]:

$$\begin{aligned} P[b' = n_{ob_i}] &= P[b' = n_{ob_i},\, b = n_{ob}] \\ &= P[b = n_{ob}] \cdot P[b' = n_{ob_i} | b = n_{ob}] \\ &= p_{bn_{ob}} \cdot p_{b'n_{ob_i}|bn_{ob}} \end{aligned}$$

where $P[b = n_{ob}] = p_{bn_{ob}}$ and $P[b' = n_{ob_i}|b = n_{ob}] = p_{b'n_{ob_i}|bn_{ob}}$. We assume that n_{ob_i} is new. This assumption is always realistic for disassembler visualizations because uniqueness of node n_{ob_i} is guaranteed by the uniqueness of the label at the beginning of the node or the uniqueness of the branch target label at the end of it. It is a worst-case assumption of reverse engineers having perfect disassembly tools working on machine code.

Let c be the number of reachable nodes of the transformed program P'. The contribution to the total entropy brought by the new reachable nodes n_{ob_i} $(i \leq 1 \leq c)$ is

$$\begin{aligned} \Delta H_{b'n_{ob_c}|bn_{ob}} &= \textstyle\sum_i^c p_{bn_{ob}} \cdot p_{b'n_{ob_i}|bn_{ob}} \cdot log_2(\frac{1}{p_{bn_{ob}} \cdot p_{b'n_{ob_i}|bn_{ob}}}) \\ &= (1 - p_{b'n_h|bn_{ob}} - p_{b'n_t|bn_{ob}}) \cdot p_{bn_{ob}} \cdot log_2(\tfrac{1}{p_{bn_{ob}}}) \\ &\quad + p_{bn_{ob}} \cdot \textstyle\sum_i^c p_{bn_{ob_i}|bn_{ob}} \cdot log_2(\frac{1}{p_{bn_{ob_i}|bn_{ob}}}) \end{aligned}$$

The contribution to the entropy brought by the two new top and bottom nodes (see Figure 1) n_h and n_t is:

$$\begin{aligned} \Delta H_{t,h} = \;& p_{bn_{ob}} \cdot p_{b'n_h|bn_{ob}} \cdot log_2(\tfrac{1}{p_{bn_{ob}} \cdot p_{b'n_h|bn_{ob}}}) \\ & + p_{bn_{ob}} \cdot p_{b'n_t|bn_{ob}} \cdot log_2(\tfrac{1}{p_{bn_{ob}} \cdot p_{b'n_t|bn_{ob}}}) \\ = \;& p_{bn_{ob}} \cdot (\textstyle\sum_{i\in\{h,t\}} p_{b'n_i|bn_{ob}} \cdot log_2(\frac{1}{p_{b'n_i|bn_{ob}}}) \\ & + (p_{b'n_h|bn_{ob}} + p_{b'n_t|bn_{ob}}) \cdot log_2(\tfrac{1}{p_{bn_{ob}}})) \end{aligned}$$

where the other probabilities are calculated in a similar way as done for $P[b' = n_{ob_i}] = P[b' = n_{ob_i}, b = n_{ob}]$ and, most importantly, the nodes are assumed to be new as well. This assumption is realistic for worst-case perfect disassembler visualization because the inserted node uniqueness is guaranteed by the uniqueness of the label at the beginning of the nodes or the uniqueness of the branch target label at the end of the nodes. The total contribution $\Delta H_{t,h,b'n_{ob_c}|bn_{ob}} = (\Delta H_{t,h} + \Delta H_{b'n_{ob_c}|bn_{ob}})$ is:

$$\Delta H_{t,h,b'n_{ob_c}|bn_{ob}} = p_{bn_{ob}} \cdot log_2(\frac{1}{p_{bn_{ob}}}) + p_{bn_{ob}} \cdot H_{P'_{n_{ob}}}$$

Before the transformation, the entropy in P: H_P was without the term:

$$\begin{aligned} p_{bn_{ob}} \cdot H_{P'_{n_{ob}}} = \;& p_{bn_{ob}} \cdot [\textstyle\sum_i^c p_{bn_{ob_i}|bn_{ob}} \cdot log_2(\frac{1}{p_{bn_{ob_i}|bn_{ob}}}) \\ & + \textstyle\sum_{i\in\{h,t\}} p_{b'n_i|bn_{ob}} \cdot log_2(\frac{1}{p_{b'n_i|bn_{ob}}})] \end{aligned}$$

This leads to the entropy of transformed program P':

$$H_{P'} = H_P + p_{bn_{ob}} \cdot H_{P'_{n_{ob}}}$$

Note that the entropy of the transformed program is given by the original one increased with the entropy of the added code weighted by the probability of its reachability. Here

$$\begin{aligned} H_{P'_{n_{ob}}} = \;& \textstyle\sum p_{bn_{ob_i}|bn_{ob}} \cdot log_2(\frac{1}{p_{bn_{ob_i}|bn_{ob}}}) + \\ & + \textstyle\sum_{i\in\{h,t\}} p_{b'n_i|bn_{ob}} \cdot log_2(\frac{1}{p_{b'n_i|bn_{ob}}}) \end{aligned}$$

[1] Obfuscation target node n_{ob} in P is as likely as either one of its sub-pieces in the transformed program P'

is the entropy of the subprogram inserted by the transformation and replacing node n_{ob} and is explicitly divided in the series coming from the switch nodes and the series coming from the stand-alone nodes. This shows that the entropy increase after *one* branch insertion transformation. By induction we prove an entropy increase after an arbitrary number of sequential branch insertion transformations introducing new nodes. ■

A. Practical quantitative consequences

In this section we consider a single step execution way of measuring entropy: it is the point of view of attackers interested more in catching basic nodes of a program that in its global behavior. There are no hypotheses of independence between the branch choices and the rest of the program to formulate this notion of entropy increase. Therefore we suppose to deal with attackers not taking care of control-flow outcomes. The theorem above shows that

$$H_{P'} = H_p + p_{bn_{ob}} \cdot H_{P'_{n_{ob}}}$$

suggesting the following guidelines regarding the application of branch insertion transformations in order to maximize $H_{P'}$:

1) Targeting frequently encountered control-free blocks of code;
2) Each instance of the obfuscation objective node should be obfuscated;
3) Each (extra) node n_{ob_i} introduced by the transformation should be brand new (e.g., differently obfuscated);
4) Splitting "enough" and performing extra jumps.

In practice, in order to protect high level language code, after an unknown opaque predicate insertion, we should apply to each alternative execution path generated (as well as to any of the newly introduced nodes) further obfuscating transformations, in order to reach a high probability of encountering blocks having a *new appearance*. In practice, when protecting local node computations, we want to keep attackers from re-recognizing the same control-free blocks during their reverse engineering activity on the same program. An easy way to achieve this is to combine two obfuscation primitives [18]: *splitting* and *outlining*: *split* the obfuscation target node and *outline* the resulting code fragments with new function names.

As indicated in the proof, worst-case disassembly visualizations (hypotheses of perfect reverse engineering tools) always satisfy the latter two tips because various control-free block labels and various control-free block branch target labels guarantee the differences among the nodes, even when two nodes perform exactly the same computations. In practice it is like providing reverse engineers with a set of boxes *always* differing at least in names. When protecting executable or assembly code, applying further transformations (e.g., like branch functions [17] or abstraction-breaking outlining), may result in a big improvement of the entropy of $H_{P'}$. The first guideline is intuitive and basically means that it is better to obfuscate computations that are frequently used. The last one interestingly explains the effectiveness of `TRUE/FALSE` type opaque predicates or techniques leaving unreachable code like the dead lumps used in [4].

The entropy considered in this section has been described using cheap control-flow transformations, that bring little performance penalty. Indeed, our measure of entropy includes also information related with execution time: there is a one-to-one correspondence between probability and execution time of a control-free block, hence if a control-free block is made slower by a transformation, then it increases its probability, modifying the distribution as well. For example if we start with an uniform distribution (maximum entropy) and make a node slower than the others, the entropy could decrease. If more costly transformation in addition to the (intentionally) minimalistic set treated in this paper were applied, further guidelines should be taken care of in order to apply the quantitative model of this section:

- Equally distributing performance overhead, not allowing any node to be unfairly "heavier" than others. Ideally, letting each node have the same probability.
- Increasing diversity among nodes. It is likely to guess that different algorithms applied to each node would tend to guarantee this. This connects to the rigorous potency definition found in [10].

The first statement above also covers the cases of transformations introducing execution paths which depend on complicated functions. Techniques that successfully confuse static disassemblers [24], though not covered by the worst-case assumption we have made, can be considered as some of the ways to introduce diversity among nodes.

B. A code obfuscating algorithm

In this section we present an application that consists in multiple simple unknown opaque predicate insertions for assembly code. The program, receiving as input code fragments provided only with their essential labels, after renaming labels and removing any comment, recognizes purely sequential blocks of code containing the first instantiation of an objective instruction and its destination register, then replicates the remaining portion of code of the block giving two alternative identical pieces based on a branch instruction conditioned by the register value. The resulting program can be transformed again with the same method, later called $split_{all}(CFG)$, either replicating every time the objective instruction is found or randomly skipping ($split_{random}(CFG)$) replication to provide different execution paths (i.e., one path intact and the other split) or identical execution paths (i.e., both paths intact or both equally split). Qualitatively speaking, the transformed program always contains extra control-free blocks that are parts of the original blocks. The critical operations recognized as such by the algorithm are every addition or subtraction (not floating points operations) and the branch condition is the comparison with a fixed threshold. In this example, the program is repeatedly fed to the algorithm as input until the CFG stabilizes, almost until the algorithm reaches a fixpoint, in order to maximize the resulting intricacy. At that point, further local randomized replications named $split_{random}(CFG)$ are done to induce code diversity (actually different pieces), but other obfuscation steps could be made like `TRUE/FALSE` types opaque predicates insertions[10] instead of unknown

```
.data
label1:  .word 0 : 24
label2:  .word 24
.text
         la   $t0, label1     #entry
         la   $t5, label2
         lw   $t5, 0($t5)
         li   $t2, 1
         sw   $t2, 0($t0)
         sw   $t2, 4($t0)
         addi $t1, $t5, -2    #critical operation
label11: lw   $t3, 0($t0)     #loop (skipped)
         lw   $t4, 4($t0)
         add  $t2, $t3, $t4   #critical operation
         sw   $t2, 8($t0)     #first block instruction
         addi $t0, $t0, 4
         addi $t1, $t1, -1    #last block instruction
         bgtz $t1, label11
         la   $a0, label1     #fibonacci exit block
         add  $a1, $zero, $t5 #critical operation
         jal label30          #jump, cannot insert if-else
         li   $v0, 10
         syscall
.data
label27: .asciiz " "
label28: .asciiz "output is:\n"
.text
label30: add  $t0, $zero, $a0 #critical operation
         add  $t1, $zero, $a1 #first block instruction
         la   $a0, label28
         li   $v0, 4          #last block instruction
         syscall
label35: lw   $a0, 0($t0)
         li   $v0, 1
         syscall
         la   $a0, label27
         li   $v0, 4
         syscall
         addi $t0, $t0, 4     #critical operation
         addi $t1, $t1, -1    #first and last block instruction
         bgtz $t1, label35
         jr   $ra
```

Table I
FIBONACCI SEQUENCE CODE

ones. The algorithm explicitly presents a subset of the defense primitives appearing in [7]: The strategies adopted by this algorithm are *duplicate* and *split* iterated according to the following recursive scheme:

$$\begin{cases} P_0 &= P \\ P_n &= split_{all}(P_{n-1}) \end{cases}$$

such that $P_n \neq P_{n-1}$ for some $n \geq 1$ and $P_{out} = split_{random}(P_{n-1})$. This sequence terminates because at each step the size of blocks is reduced and a finite number of smaller blocks is added.

Example 1: A Fibonacci computing and printing sequence has been chosen as the input example for the algorithm (Table I). Table I and II show the results (before and after the obfuscation). The transformation visibly increases instruction count by about a half of the length of the input program, besides almost maximizing cyclomatic complexity with respect to the possibilities of the algorithm. The original and new pieces are evidenced with the comment (preceded by "#") on the right column.

The entry block (7 instructions long) is executed once, the loop (6 instructions long) is executed 23 times and the exit block (2 instructions long) leading to the output subroutines also once, giving entropy:

$$\begin{aligned} H_P &= [7 \cdot 1 \cdot log_2(141/(7 \cdot 1)) + 22 \cdot 6 \cdot log_2(141/(22 \cdot 6)) + \\ &\quad 2 \cdot 1 \cdot log_2(141/(2 \cdot 1))]/141\, bits/block \\ &= (30.4 + 12.6 + 12.3)/141\, bits/block \\ &= 0.39\, bits/block \end{aligned}$$

Setting a few breakpoints (see Table II) in the obfuscated program and tracking where the program stops leads to an

```
.data
label1_93162:  .word 0 : 24
label2_93162:  .word 24
.text
               la $t0, label1_93162
               la $t5, label2_93162
               lw $t5, 0($t5)
               li $t2, 9
               sw $t2, 0($t0)
               sw $t2, 4($t0)
               addi $t1, $t5, -2
label11_93162: lw $t3, 0($t0)
               lw $t4, 4($t0)
               add $t2, $t3, $t4
               bgt $t2,297,label19_93162
     sw $t2, 8($t0)
     addi $t0, $t0, 4
     addi $t1, $t1, -1
               b label25_93162
label19_93162: sw $t2, 8($t0)
               addi $t0, $t0, 4
               bgt $t0,297,label24_93162
               addi $t1, $t1, -1
               b label25_93162
label24_93162: addi $t1, $t1, -1
label25_93162: bgtz $t1, label11_93162
               la $a0, label1_93162
               add $a1, $zero, $t5
               jal label36_93162
               li $v0, 10
               syscall
.data
label33_93162: .asciiz " "
label34_93162: .asciiz "output is:\n"
.text
label36_93162: add $t0, $zero, $a0
               bgt $t0,297,label43_93162
               add $t1, $zero, $a1
               la $a0, label34_93162
               li $v0, 4
               syscall
               b label47_93162
label43_93162: add $t1, $zero, $a1
               la $a0, label34_93162
               li $v0, 4
               syscall
label47_93162: lw $a0, 0($t0)
               li $v0, 1
               syscall
               la $a0, label33_93162
               li $v0, 4
               syscall
               addi $t0, $t0, 4
               bgt $t0,297,label57_93162
               addi $t1, $t1, -1
               b label58_93162
label57_93162: addi $t1, $t1, -1
label58_93162: bgtz $t1, label47_93162
               jr $ra
```

Table II
OBFUSCATED FIBONACCI CODE

increased entropy (about five times higher):

$$\begin{aligned} H_{P'} &= [7 \cdot 1 \cdot log_2(141/(7 \cdot 1)) + 22 \cdot 3 \cdot log_2(141/(22 \cdot 3)) + \\ &\quad 3 \cdot 6 \cdot log_2(141/(3 \cdot 6)) + 2 \cdot 16 \cdot log_2(141/(2 \cdot 16)) + \\ &\quad 1 \cdot 16 \cdot log_2(141/(1 \cdot 16)) + \\ &\quad 2 \cdot 1 \cdot log_2(141/(2 \cdot 1))]/141\, bits/block \\ &= (30.4 + 72.5 + 53.6 + 68.7 + 50.4 + 12.3)/141\, bits/block \\ &= 2.04\, bits/block \end{aligned}$$

The initial conditions of the series are the classical ones. It is not hard to evidence that the increase is valid for any initial condition.

III. MODELING PROGRAMS AS INFORMATION THEORETIC WHITE-BOX SOURCES OF TRACES

The results of this section refer to a holistic point of view of a program. For instance, a program P, as the one in Figure 1, can be modeled as a source whose alphabet is the set of all of its traces, each of which is intended as a concatenation of consecutive decision-point outcomes in the program:

$$\mathcal{A}_P = \{\gamma_1\gamma_2\gamma_3, \gamma_1\gamma_4\gamma_3\}$$

The transformed (obfuscated) program P' can be modeled as a source whose alphabet contains the new traces included after the branch insertion based transformation T.

$$\mathcal{A}_{P'} = \{\gamma_1\gamma_2\gamma_3, \gamma_1\gamma_5\gamma_6\gamma_7\gamma_3, \gamma_1\gamma_5\gamma_8\gamma_7\gamma_3\}$$

A white-box attacker observes the outcomes of a random variable trace $Tr(P)$ with distribution

$$P_{Tr(P)} = \{\ p_{Tr(P)}(tr) \in [0,1] \mid\ tr \in \mathcal{A}_{\mathcal{P}}\ \}$$

where $p_{Tr(P)}(tr)$ is the probability that the observed variable is the trace tr. Given γ_{ob} as the node undergoing a branch insertion transformation of program P, Σ_P^+ the set of finite traces of P which can be separated into two *distinct* sets given by traces subset Σ_P^u: obfuscation unaffected traces not meeting γ_{ob}, and by a subset of traces Σ_P^a meeting γ_{ob}. Since $\Sigma_{P'}^u = \Sigma_P^u$, the probability of not encountering obfuscated traces in the transformed program P' is equal to the probability of not encountering the obfuscation objective node γ_{ob} in the original program P.

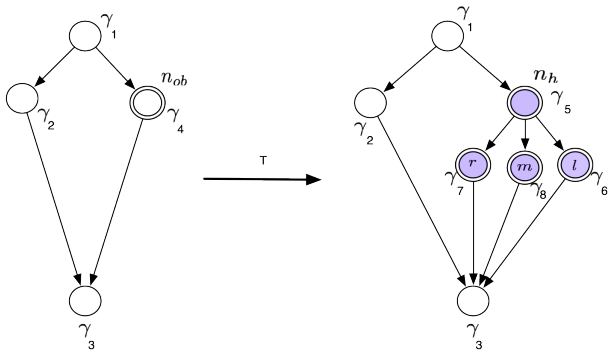

Figure 2. A 3-ways dispatcher insertion

A. Increase in traces entropy by mean of if-else statements

For the sake of simplicity, the if-else or switch-case statements are assumed to contain random conditions *independent* of the rest of the program, as if the transformed program received an extra input from external coin tosses or dice throws influencing its behavior (and confusing a white-box observer). In the calculations below, the series are summed along every path sequences, distinguishing between the affected and the unaffected ones. As already noticed, the probability of encountering or not encountering affected traces in P' is the same as in P. Moreover, in the calculation of entropy of P, the unaffected traces by the obfuscation are denoted t_u and the affected traces t_a are separated into those choosing the true path and those choosing the false path. Next theorem extends the result of Theorem 1 to the new entropy measure.

Theorem 2: The insertion of unknown binary opaque predicates independent from a program increases its traces entropy.

Proof. As a first step, we unroll every loop of the program up to its expected number of iteration. Before the insertion of *one* opaque predicate inside any of the node:

$$H_P = \sum p_{t_a} \cdot log_2\left(\frac{1}{p_{t_a}}\right) + \sum p_{t_u} \cdot log_2\left(\frac{1}{p_{t_u}}\right)$$

By inserting one opaque predicate with entropy H_{if}:

$$\begin{aligned}
H_{P'} &= \\
&\sum p_{true} \cdot p_{t_a} \cdot log_2\left(\tfrac{1}{p_{true} \cdot p_{t_a}}\right) + \\
&\sum (1 - p_{true}) \cdot p_{t_a} \cdot log_2\left(\tfrac{1}{(1-p_{true}) \cdot p_{t_a}}\right) + \\
&\sum p_{t_u} \cdot log_2\left(\tfrac{1}{p_{t_u}}\right) = \\
&p_{true} \cdot \sum p_{t_a} \cdot \left[log_2\left(\tfrac{1}{p_{true}}\right) + log_2\left(\tfrac{1}{p_{t_a}}\right)\right] + \\
&(1 - p_{true}) \cdot \sum p_{t_a} \cdot \left[log_2\left(\tfrac{1}{1-p_{true}}\right) + log_2\left(\tfrac{1}{p_{t_a}}\right)\right] + \\
&\sum p_{t_u} \cdot log_2\left(\tfrac{1}{p_{t_u}}\right) = \\
&p_{true} \cdot \sum p_{t_a} \cdot log_2\left(\tfrac{1}{p_{t_a}}\right) + p_{true} \cdot log_2\left(\tfrac{1}{p_{true}}\right) \cdot \sum p_{t_a} + \\
&(1 - p_{true}) \cdot \sum p_{t_a} \cdot log_2\left(\tfrac{1}{p_{t_a}}\right) + \\
&(1 - p_{true}) \cdot log_2\left(\tfrac{1}{1-p_{true}}\right) \cdot \sum p_{t_a} + \sum p_{t_u} \cdot log_2\left(\tfrac{1}{p_{t_u}}\right) = \\
&p_{true} \cdot \sum p_{t_a} \cdot log_2\left(\tfrac{1}{p_{t_a}}\right) + p_{true} \cdot log_2\left(\tfrac{1}{p_{true}}\right) \cdot \sum p_{t_a} + \\
&-p_{true} \cdot \sum p_{t_a} \cdot log_2\left(\tfrac{1}{p_{t_a}}\right) + \sum p_{t_a} \cdot log_2\left(\tfrac{1}{p_{t_a}}\right) + \\
&(1 - p_{true}) \cdot log_2\left(\tfrac{1}{1-p_{true}}\right) \cdot \sum p_{t_a} + \sum p_{t_u} \cdot log_2\left(\tfrac{1}{p_{t_u}}\right) = \\
&\sum p_{t_a} \cdot log_2\left(\tfrac{1}{p_{t_a}}\right) + (1 - p_{true}) \cdot log_2\left(\tfrac{1}{1-p_{true}}\right) \cdot \sum p_{t_a} + \\
&+p_{true} \cdot log_2\left(\tfrac{1}{p_{true}}\right) \cdot \sum p_{t_a} + \sum p_{t_u} \cdot log_2\left(\tfrac{1}{p_{t_u}}\right) = \\
&\sum p_{t_a} \cdot log_2\left(\tfrac{1}{p_{t_a}}\right) + \sum p_{t_u} \cdot log_2\left(\tfrac{1}{p_{t_u}}\right) + \\
&+\left[(1 - p_{true}) \cdot log_2\left(\tfrac{1}{1-p_{true}}\right) + p_{true} \cdot log_2\left(\tfrac{1}{p_{true}}\right)\right] \cdot \sum p_{t_a} = \\
&H_P + H_{if} \cdot \sum p_{t_a} = \\
&H_P + H_{if} \cdot p_a > \\
&H_P
\end{aligned}$$

Supposing that the insertion of $N > 0$ opaque predicates inside a program P, leading to program P'' brings entropy $H_{P''} > H_P$. The result above can be applied to P'' leading to program P''' with entropy $H_{P'''} > H_{P'} > H_P$. As a consequence, sequential independent unknown opaque predicates insertions increase a program traces entropy. ■

B. Increase in traces entropy by mean of switch-case statements

Figure 2 shows a (three-ways) dispatcher insertion and the relative affected and unaffected traces. The purpose of this subsection is to generalize to multi-choice switch-case statements the results of previous subsection in a more concise way, confirming the previous result through an extra alternative *path*. The single contribution to the entropy caused by *any* one the affected traces t_a is given by the contributions of the corresponding paths, denoted ct_a traces (with $c \in \{l, m, r\}$ representing the chosen path with probability p_c) $p_{ct_a} = p_{t_a} \cdot p_c$ in the transformed program:

$$\begin{aligned}
\Delta H_{P'}(ct_a) &= \textstyle\sum_c \left[p_{t_a} \cdot p_c \cdot log_2\left(\frac{1}{p_{t_a} \cdot p_c}\right)\right] \\
&= p_{t_a} \cdot \left[\textstyle\sum_c p_c \cdot log_2\left(\frac{1}{p_c}\right)\right] + p_{t_a} \cdot log_2\left(\frac{1}{p_{t_a}}\right) \cdot \textstyle\sum_c p_c \\
&= p_{t_a} \cdot \underbrace{\left[\sum_c p_c \cdot log_2\left(\frac{1}{p_c}\right)\right]}_{H_{switch}} + p_{t_a} \cdot log_2\left(\tfrac{1}{p_{t_a}}\right) \\
&= p_{t_a} \cdot H_{switch} + p_{t_a} \cdot log_2\left(\tfrac{1}{p_{t_a}}\right)
\end{aligned}$$

The series above is summed along the inserted dispatcher program sub-traces c. The series below is the final result, obtained summing the entropy given by the unaffected traces plus the contribution above summed along each of the affected program traces:

$$\begin{aligned} H_{P'} = & \sum_{t_u} p_{t_u} \cdot log_2\left(\frac{1}{p_{t_u}}\right) + \sum_{t_a} \Delta H_{P'}(ct_a) \\ = & \sum_{t_u} p_{t_u} \cdot log_2\left(\frac{1}{p_{t_u}}\right) + \sum_{t_a} p_{t_a} \cdot log_2\left(\frac{1}{p_{t_a}}\right) + \\ & H_{switch} \cdot \underbrace{\sum_{t_a} p_{t_a}}_{} \\ = & H_P + H_{switch} \cdot p_a > H_P \end{aligned}$$

As a consequence, the insertion of an arbitrary number of independent unknown dispatchers into a program increases its traces entropy.

C. Practical quantitative consequences and related works

The point of view of this work focuses on transformations that generate reachable code, which is typically examined by dynamic attackers. In this section we do not introduce an algorithm derived by the above considerations, but we rather point out existing solutions whose particular effectiveness can be explained by the the model we have introduced. The model is different from the one introduced in Section II, because it discusses the protection of a sequence of control-flow outcomes (or transitions from a node to another), rather than the computations of the nodes. If H_{if} is the entropy associated with a two-ways switch:

$$H_{P'} = H_P + H_{if} \cdot p_a > H_P$$

suggests intuitive guidelines regarding the application of opaque binary predicates, especially when considering cost based trade-offs:

1) inserting the unknown predicate to maximize p_a (for example in the entry node or the exit nodes).
2) choosing a very low bias (i.e., similar probabilities) or choosing equal left and right predicate probabilities (unbiased).

$$H_{P'} = H_P + H_{switch} \cdot p_a > H_P$$

Similarly, this suggests the following guidelines for switching:

1) inserting the unknown dispatcher to maximize the probability of encountering it p_a.
2) letting some of the cases have a slightly higher probability of execution. Or choosing equal cases probabilities (unbiased).
3) let a high number of cases in the dispatcher.

An example of unknown dispatcher is found in [23]: the first example of obfuscator can be described as a probabilistic dispatcher with $N!$ choices, essentially the orders of execution of N purely sequential methods, the other example shows a similar obfuscation but for programs including loop and conditional statement. In [14] is presented a method that transforms the entire code into a finite automaton made of a switch-case statement (probability of encountering it equals to 1) with very high number of choices (making a high entropy easy). The point of view of traces entropy should describe the effort of a reverse engineer attempting to discover the overall program behavior or to find out how rich in functionality the program during a profiling reveals to be.

D. Generalization

We have made the comfortable hypotheses of inserting branches having choices independent of the rest of the program. That has the advantage of straightforward practical quantitative advices regarding obfuscation transformation applications. But what happens to the traces entropy when we simply introduce an unknown opaque predicate based on a simple condition which is recognized as dependent on the rest of the program?

Theorem 3: Branch insertion based transformations applied to a program increase its traces entropy.

Proof. As a first step we unroll each loop of P up to their expected iteration number. The single contribution to the entropy caused by *any* one the affected traces t_a is given by the contributions of the corresponding ct_a traces ($p_{ct_a} = p_{t_a} \cdot p_{c|t_a}$) in the transformed program:

$$\begin{aligned} & \Delta H_{P'} = \\ & \sum_c \left[p_{t_a} \cdot p_{c|t_a} \cdot log_2\left(\frac{1}{p_{t_a} \cdot p_{c|t_a}}\right)\right] = \\ & p_{t_a} \cdot \left[\sum_c p_{c|t_a} \cdot log_2\left(\frac{1}{p_{c|t_a}}\right)\right] + log_2\left(\frac{1}{p_{t_a}}\right) \cdot \sum_c p_{c|t_a} \cdot p_{t_a} = \\ & p_{t_a} \cdot \left[\sum_c p_{c|t_a} \cdot log_2\left(\frac{1}{p_{c|t_a}}\right)\right] + log_2\left(\frac{1}{p_{t_a}}\right) \cdot \sum_c p_{ct_a} = \\ & p_{t_a} \cdot \underbrace{\left[\sum_c p_{c|t_a} \cdot log_2\left(\frac{1}{p_{c|t_a}}\right)\right]}_{H_{switch|t_a}} + log_2\left(\frac{1}{p_{t_a}}\right) \cdot p_{t_a} = \\ & p_{t_a} \cdot H_{switch|t_a} + p_{t_a} \cdot log_2\left(\frac{1}{p_{t_a}}\right) \end{aligned}$$

where

- $p_{c|t_a}$ is the probability of branch choice c given affected trace t_a;
- $p_{ct_a} = p_{c|t_a} \cdot p_{t_a}$ is the probability of encountering the new corresponding obfuscated trace taking branch choice c;
- $H_{switch|t_a}$ is the entropy of the new switch, given trace t_a, or, to be exact, given the prefix of t_a ending with the last choice before the newly inserted switch.

In the calculations from the third-to-last to the second-to-last line we have exploited the property $\sum_c p_{ct_a} = p_{t_a}$. The final result is obtained summing the entropy given by the unaffected traces plus the contribution above summed along each of the affected program traces:

$$\begin{aligned} & \sum_{t_u} p_{t_u} \cdot log_2(\tfrac{1}{p_{t_u}}) + \sum_{t_a} p_{t_a} \cdot log_2(\tfrac{1}{p_{t_a}}) + \sum_{t_a} p_{t_a} \cdot H_{switch|t_a} \\ & = H_P + \sum_{t_a} p_{t_a} \cdot H_{switch|t_a} \geq H_P \end{aligned}$$

By induction we can prove the increase after an arbitrary number of sequential insertions. ■

The only (counterintuitive) practical quantitative advice given by this theorem is that is preferable to target nodes which are as close as possible to the entry node, because the more conditioned are the entropies, the lower they are [9]. Therefore, further quantitative hints would depend on the particular computations inside each node and, off course, the input distributions. This theorem proves that the algorithm we have presented in Section II, as well as other obfuscation transformation algorithms introducing arbitrary unknown opaque predicates, increases traces entropy too. Particularly significant is the fact that even mere duplication of alternative

execution paths can work. In order to have approximations of increased *traces* entropy in the Fibonacci program, static analysis methods like abstract interpretation [5] can come in handy, since purely empirical methods generally cannot cover all cases generated by input distributions (such as those of a 32 bit unsigned integer in this program) or pseudo-randomness. All we say is that before the transformation, traces entropy was zero: no matter the initial conditions, the program profile would begin with the entry block, enter a loop and exit; it is not hard to find out a couple of initial conditions that produces a corresponding couple of different profiles in the transformed program, evidencing an entropy increase.

IV. Conclusions

As emerged, we believe that entropy can be a good enough measure to explain a reverse engineering deterrent to human observers, typically those employing the initial effort, while cyclomatic complexity can be good enough to explain the potency in terms of resistance to automatic deobfuscators (resilience). We have distinguished between an entropy based on a rough single step execution watchpoint, protecting local computations, and a holistic traces entropy based on an overall program behavior, protecting control-flow. We have presented an obfuscation algorithm that systematically introduces unknown binary predicates in a program containing genuine additions or subtractions. It has been repeatedly applied to a Fibonacci sequence computing program till one step before the output CFG reached a fixpoint (thus almost maximizing cyclomatic complexity with respect to the algorithm possibility), followed by the application of a similar algorithm that occasionally skips branch insertions to provide execution path diversity. The predicates could be made much more complex (for instance through comparisons between different registers or pseudorandom number generation) and the blocks could undergo further transformations, but in this work we have privileged a not too expensive approach. The essential idea is that the program entropy is a property that in both cases has been increased.

References

[1] D. Aucsmith. Tamper resistant software: an implementation. LNCS 1174, pp. 317–333, Springer 1996.

[2] D. Aucsmith and G. Graunke. Tamper resistance method and apparatus. Patent number 5892899, 1999.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18. Springer-Verlag, 2001.

[4] S.Chow, Y. Gu, H. Johnson, and V.A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. 2007.

[5] P. Cousot and M. Monerau. Probabilistic abstract interpretation. *ESOP2012*, 2012.

[6] C. Collberg, J. Davidson, R. Giacobazzi, Y. Xiang Gu, A. Herzberg, and F.-Y. Wang. Toward digital asset protection. *IEEE Intelligent Systems*, 26(6):8–13, 2011.

[7] C. Collberg and J. Nagra. *Surreptitious Software*. Addison-Wesley Pearson Education, 2010.

[8] C. Collberg, C. D. Thomborson, and D. Low. Manufactoring cheap, resilient, and stealthy opaque constructs. In *Proc. of Conf. Record of the 25st ACM Symp. on Principles of Programming Languages* (*POPL '98*), pages 184–196. ACM Press, 1998.

[9] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, 2006.

[10] M. Dalla Preda and R. Giacobazzi. Semantic-based Code Obfuscation by Abstract Interpretation. *Journal of Computer Security*, 17(6):855-908, 2009.

[11] B. De Sutter, B. De Bus, K. De Bosschere, B. Preneel, B. Anckaert, and M. Madou. Program obfuscation: A quantitative approach. In *QoP '07: Proceedings of the 2007 ACM workshop on Quality of protection*, 2007.

[12] R. Giacobazzi. Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In *Proc. of The 6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM'08)*, pages 7–20. IEEE Press., 2008.

[13] R. Giacobazzi, N.D. Jones, and I. Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In *Proceedings of the ACM Symposium on Partial Evaluation and Program Manipulation* (*PEPM '12*). ACM Press, 2012.

[14] Y. Gu, S. Chow, and H. Johnson. Tamper resistant software: control flow encoding. *Patent filed under the Patent Cooperation Treaty*, PCT/CA00/00943, 2000.

[15] A. Klenke. Probability Theory: A Comprehensive Course. Springer (2007)

[16] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.

[17] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. *10th ACM Conference on Computer and Communications Security (CCS)*, E86-A(1), 2003.

[18] D. Low, C. Collberg C. Thomborson. A taxonomy of obfuscating transformation, technical report 148. Technical report, Department of Computer Science The University of Auckland.

[19] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 4:308–320, 1976.

[20] S. S. Muchnick and N. D. Jones. *Program Flow Analysis: theory and applications*. Prentice-Hall, 1981.

[21] F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding – A survey. *Proc. of the IEEE*, 87(7):1062–1078, 1999.

[22] C. E. Shannon. A Mathematical Theory of Communication. Bell System Tech. J., vol. 27, 1948.

[23] T. Tabata, T. Toyofuku, and K. Sakurai. Program obfuscation scheme using random numbers to complicate control flow. *LNCS 3823*: pp. 1–10, 2005.

[24] F. Valeur, C. Kruegel, W. Robertson, and G. Vigna. Static disassembly of obfuscated binaries. *SSYM 2004 Proceedings of the 13th Conference on Usenix Security Symposium*, pages 18–18, 2004.

[25] M. Nakamura, A. Monden, T. Itoh,K. Matsumoto,Y. Kanzaki and H. Satoh. Queue-based Cost Evaluation of Mental Simulation Process in Program Comprehension *Proceedings of the Ninth IEEE International Software Metrics Symposium (METRICSâĂŹ03)*,1530-1435/03, 2003