Automatic Generation of Opaque Constants Based on the K-Clique Problem for Resilient Data Obfuscation

Roberto Tiella
FBK
Trento, Italy
Email: tiella@fbk.eu

Mariano Ceccato
FBK
Trento, Italy
Email: ceccato@fbk.eu

***Abstract*—Data obfuscations are program transformations used to complicate program understanding and conceal actual values of program variables. The possibility to hide constant values is a basic building block of several obfuscation techniques. For example, in XOR Masking a constant mask is used to encode data, but this mask must be hidden too, in order to keep the obfuscation resilient to attacks. In this paper, we present a novel technique based on the k-clique problem, which is known to be NP-complete, to generate opaque constants, i.e. values that are difficult to guess by static analysis. In our experimental assessment we show that our opaque constants are computationally cheap to generate, both at obfuscation time and at runtime. Moreover, due to the NP-completeness of the k-clique problem, our opaque constants can be proven to be hard to attack with state-of-the-art static analysis tools.**

I. Introduction

Programs often contain sensitive data, e.g., license numbers or decryption keys, that could be extracted and stolen in case the code is scanned or analysed by malicious users (man-at-the-end attack model [1]). Among the possible protection strategies that can be applied to limit malicious program understanding, data obfuscation aims at complicating data extraction by changing data representation and use.

Many approaches for obfuscating the program *control flow* have been defined and are quite largely adopted in practice [2]. However, even if the control flow is extremely difficult to understand, when values are stored in clear in the program binary, they can be easily attacked, because control flow obfuscation does not impede data extraction. In fact, clear data can be read from the binary program before execution. For this reason, *data* obfuscation should complement *control* obfuscation in protecting against malicious reverse engineering. Though, we want to stress here the fact that, even if in the following we will show source code just after the application of data obfuscation, we are assuming that control flow obfuscation techniques will be applied to mangle the resulting source code.

While obfuscation transformations are quite effective in turning programs hard to *understand* by human attackers [3], they should be also hard to *analyse* by automated tools. *Opaque constants* are program constants, whose value is opaque for a static analysis tool, because the value is computed at runtime by the program (once needed), but the value is very difficult to recover by a static analysis tool. Moser et al. [4] presented an approach based on opaque constants to block static analysis tools. They used opaque constants as a building block to implement advanced obfuscation algorithms.

The approach proposed by Moser et al. relies on the fact that a static analysis tool (i.e., the attacker) would have to solve a NP-complete problem, namely a 3SAT problem, to guess the values of opaque constants. However, for the obfuscation transformation to be sound (in Section III we list obfuscation transformation's requirements, soundness is among them), the same problem should be also faced by the automatic obfuscating tool (i.e., the defender). Thus, the opaqueness of data can not be too high, otherwise the obfuscation tool would be also unable to terminate the obfuscation task.

In this paper, we propose a novel approach to opaque constants, such that they are difficult to guess with static analysis tools, but also easy to generate for an automatic obfuscation tool. Similarly to Moser et al., we also start from a small 3SAT problem, that is supposed to be solved at obfuscation time. However, this problem is mapped to a k-clique problem of larger size, that a static analysis tool should solve to identify the opaque value. The effectiveness of our approach relies on an asymmetry of the effort needed at obfuscation time and at attack time.

The novel contributions of this paper are:

- The explicit identification of the requirements for effective opaque constants;
- The presentation of a novel approach for opaque constants based on the k-clique problem; and
- The empirical assessment of our opaque constants against a state-of-the-art static analysis tool, and its comparison with the previous approach by Moser et al.

The paper is structured as follows. Background on data obfuscation is covered in Section II. The problem of strengthening obfuscations by opaque constants in exposed in Section III. Section IV presents our approach based on the k-clique problem which is empirically evaluated in Section V. Section VI comments related work and, lastly, Section VII summaries conclusions and future work.

II. Background

Obfuscation [5] denotes program transformation techniques aiming at turning a program into another one more hard to understand and thus to attack, while preserving execution semantic. In particular, *XOR Masking* and *Residue Number Coding* are two program transformation approaches meant to complicate the representation of data to make them harder to comprehend.

A. XOR Masking

XOR Masking is a quite frequently [6] used data obfuscation transformation based on the bitwise XOR operator. Variants of XOR Masking are known to be employed by practitioners and by malware [7]. This obfuscation consists in encoding a clear value by computing its bitwise XOR (denoted with ˆ) with an integer constant (the mask) p: $e(x) = x \hat{} p$.

From the property of the XOR operator that $(x \hat{} p) \hat{} p = x$, it follows that the clear value can be recovered using the decoding function that is $e(\cdot)$ itself.

Figure 1 contains snippets of code before and after having applied the XOR Masking obfuscation with $p = 12$. The values of program variables a, b and x are stored encoded in memory, and they are decoded only where the clear values are needed. In the third line, a and b are decoded to use their values in a sum, and the result is encoded before being assigned to x. The program variable x is decoded in the last line, to be printed.

```
int a = 5;              int a = 9; // 9 = 5^12
int b = 8;              int b = 4; // 4 = 8^12
int x = a+b;    →       int x = ((a^12)+(b^12))^12;
...                     ...
printf("%d\n",x);       printf("%d\n",x^12);
```

Fig. 1. Example of XOR Masking.

B. Residue Number Coding

Residue Number Coding (RNC for short) [8] is an encoding on integers which is homomorphic for both the sum (and thus the subtraction) and the product, i.e., it does not require to decode back encoded values to perform computations.

The definition of RNC is based on modular arithmetic and, in what follows, $[x]_m$ will denote the *congruence class modulo m represented by x*, that is to say the set of integers y that are equivalent to x modulo m, i.e. $y = x + km$ for some $k \in \mathbb{Z}$ (also written as $y \equiv x (\text{mod } m)$). Any element $y \in [x]_m$ can be chosen as class representant, i.e. if $y \in [x]_m$ then $[y]_m = [x]_m$.

To apply RNC, u modules $m_1, m_2, ..., m_u \in \mathbb{Z}$ should be chosen such that they are pairwise co-prime (i.e., $gcd(m_i, m_j) = 1$, for $i \neq j$). Having the product of modules $n = m_1 \cdot m_2 \cdot ... \cdot m_u$, RNC consists in encoding a value $x \in [0, n-1]$ into u components in the following way:

$$e(x) = < [x]_{m_1}, [x]_{m_2}, ..., [x]_{m_u} >$$

The function e is invertible due to the fact that, given $< y_1, y_2, ..., y_u >$ with $y_i \in \mathbb{Z}$, the Chinese Remainder Theorem ensure existence and unicity (modulo n) of x satisfying the following system of congruences:

$$\begin{cases} x \equiv y_1 (\text{mod } m_1) \\ ... \\ x \equiv y_u (\text{mod } m_u) \end{cases}$$

Furthermore x can be computed by using Euclid's extended algorithm for the gcd [9]. Operations in the encoded domain are defined component by component:

$$< [x_1]_{m_1}, [x_2]_{m_2}, ... > + < [y_1]_{m_1}, [y_2]_{m_2}, ... > = < [x_1 + y_1]_{m_1}, [x_2 + y_2]_{m_2}, ... >$$

$$< [x_1]_{m_1}, [x_2]_{m_2}, ... > * < [y_1]_{m_1}, [y_2]_{m_2}, ... > = < [x_1 * y_1]_{m_1}, [x_2 * y_2]_{m_2}, ... >$$

Figure 2 shows an example of RNC encoding of program variables x, y, z and w using the modules $m_1 = 31$ and $m_2 = 37$. Each original value is split in two encoded values (e.g., x is split in $x1$ and $x2$) because two modules are used as encoding base. At (original) lines 1 and 2, constants in the right hand side of the assignments are replaced by their encoded component, 12 is replaced by <1965,1973>, while 7 is replaced by <1433,2634>. To increase obscurity of values, random representatives are chosen for constant encoded values.

Sum and multiplication at lines 3 and 4, are performed component-wise in the encoded domain and assigned to z and w. In the last statements, z and w are decoded (using the support function `d`) before being printed. We can note that the clear values of x and y are never revealed in clear by this program.

```
                            // 1965 % 31 = 12
                            int x1 = 1965;
                            // 1973 % 37 = 12
                            int x2 = 1973;

1: int x = 12;
2: int y = 7;               // 1433 % 31 = 7
                            int y1 = 1433;
3: int z = x+y;             // 2634 % 37 = 7
4: int w = x*y;       →     int y2 = 2634;

                            int z1 = x1+y1;
5: printf(                  int z2 = x2+y2;
  "z=%d, w=%d\n",
   z, w);                   int w1 = x1*y1;
                            int w2 = x2*y2;

                            printf("z=%d, w=%d\n",
                                   d(z1,z2),d(w1,w2));
```

Fig. 2. Example of encoding integer variables using Residue Number Coding.

III. Problem Definition

Data encoded with the state-of-the-art XOR Masking and RNC are vulnerable to attacks based on static analysis. In fact,

the mask p used in XOR Masking and the modules m_1, m_2, ..., m_u used in RNC are static constants that, once extracted from the code, can be used to decode and obtain the clear value of obfuscated variables. A way to make these obfuscation techniques more resistant to static analysis attacks, is to turn these constants into different software entities that are harder to analyse, by substituting them with *opaque constants* [10]. A constant is opaque when its value (known at obfuscation time) is removed from the code and it is computed at runtime in a way that is hard to guess by analysing the obfuscated program. Figure 3 suggests how to improve the obfuscation in Figure 1 by using a function `oc_12` that compute the opaque constant 12.

A. Attack Model

Similarly to what assumed by related work [4], [10], in this paper we also assume that an attacker has full access to the program and that the attacker can deploy any static analysis tool and algorithm on the program. Relevant examples of tools are IDA-Pro and KLEE. The attacker objective is to extract sensitive values from the program, those values that are meant to be obfuscated.

```
int a = 9; // 9 = 5^12
int b = 4; // 4 = 8^12
int x = ((a^oc_12())+(b^oc_12()))^oc_12();
...
printf("%d\n",x^oc_12());
```

Fig. 3. Example of XOR Masking with opaque constants.

B. Opaque Constant Requirements

The first requirement descends directly by Collberg's definition of obfuscation transformation [5]: the substitution of a constant with its opaque equivalent must produce a program that behaves as the original one on valid inputs, i.e.,

> ***Requirement Req***$_1$*: The opaque constant transformation must be sound.*

The second requirement for an opaque constant is that it should make the obfuscation resilient and difficult to break. *Resilience* is defined as a measure of how well a transformation holds up under attack by an automatic deobfuscator [10]. Attackers in fact might adopt tools to perform malicious reverse engineering and try to extract decoded values from a program. For example, if the opaque value is computed starting from constant values, it could be easily recovered using static analysis techniques such as constant propagation. Thus, the opaque value should be computed starting from random values or program inputs. Moreover, recovering the opaque constant should require the attacker to solve a known hard problem, i.e. a NP-hard problem. So our second requirement is the following:

> ***Requirement Req***$_2$*: Guessing the opaque constant with static analysis should be hard.*

Conversely, from the defender point of view, the obfuscation should be computationally cheap to apply. The obfuscation should be based on a problem that, despite it is *hard* to solve by the attacker, it should be *easy* to construct and verify by the obfuscating tool. To assess that the obfuscation is correctly applied, the obfuscating tool should not solve the same problem as the attacker. Thus, the third requirement is the following:

> ***Requirement Req***$_3$*: Constructing the opaque constant at obfuscation time should not be hard.*

Additional code is inserted to a program, to compute opaque constants at execution time. When this additional code is complex, the obfuscated code might suffer sensible runtime overhead and performance degradation. Despite different execution contexts might pose different constraints to execution time, obfuscation should be in general lightweight and it should not impact too much the program execution speed. For example, when needed, an opaque value should be computed in polynomial time. Thus, the last requirement is:

> ***Requirement Req***$_4$*: Computing the opaque constant at execution time should be fast.*

C. Opaque Constant based on 3SAT

Moser et al. [4] proposed a solution to opaque constants based on the 3SAT problem. This solution is meant to provide resilient opaque constants, because an attacker should solve a hard instance of 3SAT problem in order to identify the opaque values and break obfuscation. Here, we summarize the approach proposed by Moser et al.

A 3SAT problem is defined as in the following. Let φ be a propositional formula in conjunctive normal form with the n propositional variables[1] $\{v_1, v_2, ..., v_n\}$ and with k 3-literals clauses:

$$\varphi = \bigwedge_{i=1,...,k} \alpha_{i,1} \vee \alpha_{i,2} \vee \alpha_{i,3} \text{ with } \alpha_{i,j} \in \{v_j, \neg v_j | j = 1, ..., n\} \quad (1)$$

The 3SAT problem consists in identifying the a truth assignment for variables $v_1, v_2, ..., v_n$ such that φ is satisfied (i.e., φ evaluates to *true*). An example of such propositional formula in the propositional variables v_1, v_2 and v_3 is:

$$(\neg v_1 \vee \neg v_2 \vee \neg v_3) \wedge (\neg v_1 \vee v_2 \vee \neg v_3) \wedge (v_1 \vee \neg v_2 \vee v_3) \wedge (v_1 \vee v_2 \vee v_3) \quad (2)$$

The intuition of the approach proposed by Moser et al. is that an attack based on static analysis would fall in solving a hard 3SAT problem.

[1]The term *propositional variable* is used to distinguish Boolean variables used in the 3SAT propositional formula from *program variables* subject to data obfuscation.

Figure 4 sketches the implementation of this approach. They reuse a heuristic proposed by Selman et al. [11] (discussed later in Section IV-A) to automatically generate a hard unsatisfiable propositional 3SAT formula φ. This formula is then encoded as three arrays of integer pointers, namely $a1$, $a2$ and $a3$. Each element of these three arrays is initialized (lines 4 to 6) as a pointer to an integer program variable, namely program variables v_1, v_2, ..., v_n or their negation nv_1, nv_2, ... nv_n to reflect literals $\alpha_{i,j}$ in the formula. For example, in lines 4-6, $a1[0]$,$a2[0]$, and $a3[0]$ are initialized respectively to point to $nv1$, $nv2$, and $nv3$ to encode the first clause $\neg v_1 \vee \neg v_2 \vee \neg v_3$ of formula (2).

At lines 8-9, each integer program variable v_i is initialized randomly to *false* or *true*, and the corresponding negated program variable nv_i is initialized to its negation, i.e. to *true* or *false*, respectively.

The computation of one bit of the opaque constant starts at line 10, by initializing varaible phi to *true*. Then, in the for loop, each clause is considered separately. The j-th clause is evaluated in the *if* condition at line 12. Since the formula is in conjunctive normal form, if at least one clause is *false*, the whole formula is *false* and the loop stops with the *break* statement.

However, since the formula is unsatisfiable by construction, the clauses can not be all *true* and the Boolean value of phi will be always *false* at the end of the iteration, regardless of the value of random program variables v_i and nv_i. Thus the opaque bit assigned at line 19 will be always 0. Swapping line 19 with line 21 generates an opaque 1-valued bit.

```
1  int v1, v2, ..., vn;
2  int nv1, nv2, ..., nvn;
3
4  int *a1[] = { &nv1, &nv1, ... };
5  int *a2[] = { &nv2, &v2, ... };
6  int *a3[] = { &nv3, &nv3, ... };
7
8  v1=rnd(FALSE,TRUE); nv1=!v1;
9  ...
10 int phi = TRUE;
11 for (int j=0; j<k; j++) {
12   if (!*a1[j] && !*a2[j] && !*a3[j]) {
13     phi = FALSE;
14     break;
15   }
16 }
17
18 if (!phi) {
19   opaque_bit = 0;
20 } else {
21   opaque_bit = 1;
22 }
```

Fig. 4. Generation of a single bit of the opaque constant based on a 3SAT problem by Moser et al.

D. Considerations about Requirements

According to the arguments of Moser at al., opaque constants based on 3SAT meet requirement Req_2, because their value is hard to guess statically. In fact, the opaque value is computed starting from random values, that are not available to static analysis. Thus, to guess the opaque value, symbolic execution requires to elaborate a two set of assignments to program variables v_i and nv_i, one to makes phi *true* and the other to makes it *false*. However, this would mean to solve the 3SAT formula, and detect if it is satisfiable or not, a problem known to be hard [12], that can be made harder and harder by tuning the size of the propositional formula.

Additionally, Moser at al. shown that our requirement Req_4 is also met. In fact, the execution of the code in Figure 4 to compute an opaque bit at runtime is quite fast (linear in the number of clauses) and it does not impact too much the program execution time.

However, this approach suffers a major drawback, it does not simultaneously meet requirements Req_1, Req_2 and requirement Req_3. In fact, to meet Req_1, the obfuscating tool should check that the randomly generated 3SAT formula is actually unsatisfiable to be sure that the opaque value is constant for *any* random assignment of program variables v_i and nv_i. As such, the defender is supposed to solve (at obfuscation time) the same 3SAT problem that the attacker also needs to solve (at attack time) to break the obfuscation. Thus, a 3SAT formula can not be too large and too hard to solve, otherwise it would not be tractable at obfuscation time (Req_3). A not very complex 3SAT formula implies opaque constants with limited resilience (Req_2).

In the next section, we will present our approach based on k-clique, to craft resilient and manageable opaque constants. We will show how our solution addresses all of our requirements, and requirement Req_3 in particular, missed by the 3SAT approach.

IV. Proposed Approach

Our opaque constants are based on the k-clique problem, a problem known to be NP-hard [12]. We will show how our solution meets all of the requirements for resilient opaque constants and overcome the limitations of previous work.

Intuitively, we adopt a reduction transformation to turn a propositional formula in conjunctive normal form into a graph. The reduction is defined such that the graph contains a clique of size k if and only if the formula is satisfiable. Eventually, we compute the opaque constant based on some properties of the graph, such that, to guess the opaque value, a static analysis tool would have to solve a hard k-clique problem.

Intuitively, the k-clique problem is defined as the decision whether an arbitrary undirected graph contains as subgraph of k vertexes all pairwise connected. More formally, given an undirected graph $G = (V, E)$ and an integer k (with $k \leq |V|$), the k-clique problem consists in deciding if the graph G contains a *clique* of size k (or higher). A clique of size k is a subset V' of the graph vertexes ($V' \subseteq V$) of size at least k ($|V'| \geq k$), in which all the pairs of vertexes in V' are connected by an edge in E from the original graph G.

A. Generation of a Hard 3SAT Problem

Our approach starts from an unsatisfiable propositional formula for 3SAT problem. To generate a hard instance of

the 3SAT problem, we adopt the guidelines proposed by Selman et al. [11] derived by their empirical study about random sampling 3SAT formulas. They studied 3SAT formulas generated using the *fixed clause-length model*, i.e. a model where each clause is produced by randomly choosing a set of 3 variables from the set of n available, and negating each with probability 0.5. They found that the hardest area for satisfiability is near the point where 50% of the formulas are satisfiable, and when the ratio between the number of clauses k and the number of propositional variables n is between 4.25 and 4.55. In our approach, we adopt an average ratio k/n of 4.3, i.e. our formulas will have n propositional variables and $k = \lfloor 4.3 \cdot n \rfloor$ clauses.

We run SAT solver on this random 3SAT formula to check that it is unsatisfiable. If the check fails, we discard the formula and we generate a new one, until the check with the SAT solver passes.

This process is an example of a Bernoulli trial. The expected number of times the check has to be repeated before the first success is $1/p$, where p is the probability of success. We will assessed this probability and the time required to perform the check in the evaluation part of the present work (see Section V) but we can anticipate that the total time required to generate a 3SAT unsatisfiable formula in up to 200 variables is less than a second with a very high probability.

Finally, the 3SAT unsatisfiable problem that we found is turned into a k-clique problem as described in the following.

B. Reduction of a 3SAT Problem to a k-clique Problem

A 3SAT problem can be reduced[2] to a k-clique problem in this way. Let's assume to have a 3SAT formula φ in n variables consisting in k clauses as in Equation (1). We construct a graph $G_\varphi = (V, E)$, whose vertexes V and edges E are defined according, respectively, to Equation (3) and Equation (4).

$$V = \{(i, \alpha_{i,1}), (i, \alpha_{i,2}), (i, \alpha_{i,3}) | i = 1, ..., k\} \tag{3}$$

$$E = \begin{array}{l} \{(i_1, \alpha_{i_1,j_1}), (i_2, \alpha_{i_2,j_2})) \mid \\ i_1 \neq i_2 \text{ and } \alpha_{i_1,j_1} \wedge \alpha_{i_2,j_2} \text{ satisfiable}\} \end{array} \tag{4}$$

The vertex set V contains a distinct vertex for each occurrence $\alpha_{i,k}$ of a literal in the clause i. The edge set E contains an edge for every pair of literals belonging to two different clauses and so that they are jointly satisfiable, i.e. if one is not the logical negation of the other.

Figure 5 shows the graph corresponding to the example propositional formula in Equation (2). In the figure, a minus sign '-' stands for the logical negation, for example the literal $\neg v_1$ is written $-v_1$. Nodes $(1, \neg v_2)$ and $(2, v_2)$ are not connected because v_2 and $\neg v_2$ can not be jointly satisfiable, while node $(1, \neg v_2)$ and node $(2, \neg v_1)$ are connected because $\neg v_2$ and $\neg v_1$ can be jointly satisfied (and are present in two different clauses, namely 1 and 2).

[2]Here we intend the reduction as proposed by Karp[13]. Informally, a *reduction* is the transformation of a decision problem into another by means of an algorithm that executes in polynomial time.

By construction, the 3SAT formula φ with k clauses is satisfiable if and only if the graph G_φ contains a k-clique. In fact, if the graph contains a k-clique, vertexes in the clique refer to literals that can be assigned to *true* without resulting in contradiction (all nodes in a clique are pairwise connected and by construction being connected means being jointly satisfiable). Literals mentioned in clique's vertexes belong to different k clauses as by definition two vertexes are connected only if they belong to different clauses. Thus the truth assignment satisfies all the k clauses, i.e. it satisfies φ. On the other hand, if there exists an assignments that satisfies φ, it means that for each clause i at least a literal α_{i,j_i} is true and the set $\bar{V} = \{(i, \alpha_{i,j_i}), i = 1, ..., k\}$ is a k-clique. In fact, if $(i_1, \alpha_{i_1,j_{i_1}}), (i_2, \alpha_{i_2,j_{i_2}}) \in \bar{V}$ with $i_1 \neq i_2$, it means that $\alpha_{i_1,j_{i_1}} \wedge \alpha_{i_2,j_{i_2}}$ is satisfiable thus $(i_1, \alpha_{i_1,j_{i_1}})$ and $(i_2, \alpha_{i_2,j_{i_2}})$ are connected.

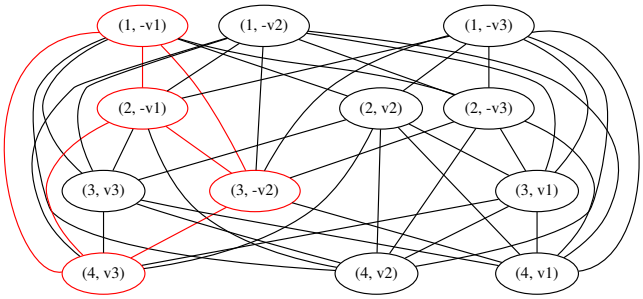

Fig. 5. Graph obtained by means of the Karp's reduction of a 3SAT problem to a k-clique problem for the formula in Example 2.

In our running example, the propositional formula in Equation (2) has 4 clauses and it is satisfiable. So the graph in Figure 5 should contain a clique of size 4. In fact, nodes $\{(1, \neg v_1), (2, \neg v_1), (3, \neg v_2), (4, v_3)\}$ form a 4-clique in the graph. They correspond to the assignment:

$$v_1 = False \qquad v_2 = False \qquad v_3 = True$$

Considering how nodes are generated starting from the propositional formula (see Equation (3)), starting from a 3SAT formula φ with k clauses, the resulting graph G_φ will have $3k$ nodes.

C. Opaque Constants based on k-clique

Our approach leverages the NP-hardness of the k-clique problem for forging resilient opaque constants. The integer value c to be turned into an opaque constant is split into n_b-bits, $c = b_0 b_1 ... b_{n_b - 1}$. To encode the bit b_i as a k-clique problem, first of all we generate the propositional formula φ_i with k clauses, corresponding to a hard 3SAT problem and we reduce it to the graph $G_i = (V_i, E_i)$ as described earlier. The propositional formula φ_i is generated such that it is unsatisfiable, i.e. it is *false* for each Boolean assignment of its propositional variables. Thus the graph $G_i = (V_i, E_i)$ contains no clique of size k, i.e. any subset of k vertexes will not be a clique.

Figure 6 shows the snippet that generates a single bit of the constant c. The function call at Line 13 randomly generates a

set of vertexes. The `generate_subset(idx,n,s)` randomly assigns elements from the set $\{0, 1, 2, ..., s-1\}$ to the vector idx of length n. In the actual obfuscation, the function `generate_subset(v,n,m)` is inlined. The nested loops at Line 15 and Line 16 verify whether the subgraph induced by the set of vertexes is a clique or not. If it is not, which is always the case by construction, the bit is set to the required value, 0 in this example (Line 18). The chunk of code is repeated for all n_b.

Figure 3 suggests how to improve the resilience XOR Masking shown in Figure 1 with an opaque constant. Each use of the constant 12 was replaced by a call to the function `oc_12` generated by our approach.

```
1   int gm [][51] = {
2     { 0,0,...,1,1,1,0,1,1,},
3       ...
4     { ... }
5   };
6
7   int phi = TRUE;
8   int i,j;
9   int n = 17;
10  int s = 3*n;
11  int idx[n*sizeof(int)];
12
13  generate_subset(idx,n,s);
14
15  for (i=0; i<n-1; i++) {
16    for (j=i+1; j<n-1; j++) {
17      if (!gm[idx[i]][idx[j]]) {
18        phi=FALSE;
19        break;
20      }
21    }
22  }
23
24 if (!phi) {
25  opaque_bit = 0;
26 } else {
27  opaque_bit = 1;
28 }
```

Fig. 6. Generation of a single bit of the opaque constant based on our k-clique problem based approach.

D. Resilience of the k-clique based approach

Our approach meets all the requirements that we identified for resilient opaque constants.

Requirement Req_4 imposes a fast computation of the opaque constant when the obfuscated program is executed. The computation of our opaque constants requires a quite short time (i.e., polynomial time). The algorithm for the computation of an opaque bit (see in Figure 6) checks if a random set of k vertexes is a clique. The algorithm verifies if the $k(k-1)/2$ vertexes needed by the clique are present in the graph. However, this algorithm can stop at the first edge that does not satisfy the condition.

Our opaque constants are based on a hard problem (the k-clique problem) and a static analysis tool should solve it to identify the value of the constant. We can easily make static analysis harder and harder (requirement Req_2) by increasing the size of the problem with more and more propositional variables.

The time required to construct the opaque constant at obfuscation time is dominated by the time needed to check if the random propositional formula (to be used for generating the graph) is satisfiable or unsatisfiable. In fact, only unsatisfiable formulas should be used. This means that, to check that the correct (unsatisfiable) propositional formula is chosen, the obfuscation tool has to solve the corresponding 3SAT problem. For this reason, the propositional formula can not be too complex. Our approach meets requirement Req_3 by limiting the complexity of the propositional formula.

In the approach by Moser et al., the obfuscation tool and the attacker should solve the same 3SAT problem, so requirement Req_2 (hard for the attacker) and requirement Req_3 (easy for the obfuscation tool) can not be meet at the same time. Conversely, our approach fully meets these two requirements, because the obfuscation tool has to solve a small 3SAT problem (Req_3) while the attacker will face a bigger k-clique problem (Req_2).

Let's quantify the asymmetry of the problems. The obfuscation tool has to check a 3SAT formula φ in n variables and $k = \lfloor 4.3 \cdot n \rfloor$ clauses. Instead, the attacker as to solve a k-clique problem on a graph with $3k$ nodes. For example, a 3SAT formula in 4 variables with $17 = \lfloor 4.3 \cdot 4 \rfloor$ clauses generates a 17-clique problem for a graph G whit 51 nodes.

Considering that no polynomial algorithm is known to solve these problems, a linear size expansion from the 3SAT to the k-clique would cause an exponential explosion of the time required to solve the lager (k-clique) problem with respect to the time required to solve the smaller (3SAT) problem.

In our evaluation (in Section V), we will show how a state-of-the-art static analysis tool behaves differently when analysing the two problems.

V. Empirical Evaluation

We performed an empirical evaluation of the proposed approach, to validate the execution time overhead involved with the usage of opaque constants and their strength against static analysis.

A. Research Questions and Variable Selection

We formalized our evaluation goals in the following research questions:

- **RQ_1:** How long does a state-of-the-art SAT solver take to check satisfiability of a 3SAT formula?
- **RQ_2:** How long does an obfuscated program take to compute the value of an opaque constant based on k-clique at runtime?
- **RQ_3:** What is the resilience of opaque constants based on 3SAT?
- **RQ_4:** What is the resilience of opaque constants based k-clique?

This first research question aims at assessing the computational effort required by the defender to generate a 3SAT problem. Later, this effort is compared with the one required by the attacker to solve the relative k-clique problem. The

second question is intended to quantify the performance degradation due to program obfuscation and study how long it takes to compute an opaque constant at runtime. Then, we are interested in comparing the effectiveness of our opaque constants with the existing approach based on 3SAT. Thus the third research question is formulated on the resilience of opaque constants based on 3SAT while the fourth research question is specifically on the resilience of opaque constants based on k-clique.

To answer these research questions, we will consider these metrics on a number of experiments:

- **ETIME**: Execution time for a program, the user time reported by the Linux's tool `time` and converted in seconds.
- **NCLS**: Number of clauses in a 3SAT problem.
- **NVARS**: Number of propositional variables in a 3SAT problem.
- **NVGR**: Number of vertexes in a graph.
- **KCLQ**: Size of a k-clique, i.e. `KCLQ`=k.
- **PSAT**: Fraction of satisfiable problems in a given set of 3SAT problems.

B. RQ_1: Time to Verify 3SAT

In the experiment we measure how long it takes to Yices[3], a state-of-the-art SMT solver, to check if a 3SAT formula in NVARS propositional variables and with NCLS = $4.3 \cdot$NVARS is satisfiable. For NVARS running from 50 to 350, we generate 100 random 3SAT formulas and for each formula we execute Yices to check if it is satisfiable or not. We collect execution time from the execution log Yices produces when the tool is run with the flag `(show-stats)`. We take NVARS $\geq$ 50 to avoid time measures be shorter than the measurement accuracy. Table I show aggregated statistics for collected data. Mean of ETIME ranges from $1.9 \cdot 10^{-4}$ for NVARS $= 50$ to 837.7 for NVARS $= 350$. PSAT ratio estimates the probability of generating a satisfiable 3SAT problem and ranges from 0.60 for 50 variables to 0.30 for 350 variables.

TABLE I
SATISFIABILITY RATIO (PSAT), MEAN AND SD OF EXECUTION TIME FOR YICES SOLVING RANDOMS 3SAT PROBLEMS IN NVARS VARIABLES.

NVARS	PSAT	MEAN(ETIME)	SD(ETIME)
50	0.60	0.00013	0.00019
100	0.51	0.00176	0.00143
150	0.49	0.01134	0.00645
200	0.45	0.09320	0.06800
250	0.37	1.09245	0.74141
300	0.37	25.42116	26.68942
350	0.30	828.21444	837.64963

Based on the empirical evidence, we can answer to RQ_1 in this way:

> *The time a state-of-the-art SAT solver takes to decide a 3SAT formula with a number of variables NVARS less than 200 is negligible.*

[3]http://yices.csl.sri.com/

C. RQ_2: Time to Compute Opaque Values at Runtime

In this second experiment we measure how long an obfuscated program takes to compute the value of an opaque constant based on k-clique.

We base this empirical assessment on a program that just calls 1,000,000 times a function f that computes an opaque constant value of 16 bits. The experiment is repeated with a increasing number of propositional variables NVARS. We start with NVARS = 4 and we increase this value in steps of size 4, until 40 propositional variables. For each value of NVARS, 10 different random 3SAT formulas have been generated. To minimize random error, the measurement of the execution time ETIME is repeated 25 times. Thus, in total, we collect 250 measurements of ETIME for each value of NVARS.

Figure 7 reports the boxplot of the execution time ETIME for increasing values of NVARS. It can be noted that the computation of 1,000,000 opaque constants on average takes from 4.13 seconds for formulas with 4 propositional variables up to 4.48 seconds for formulas with 40 propositional variables.

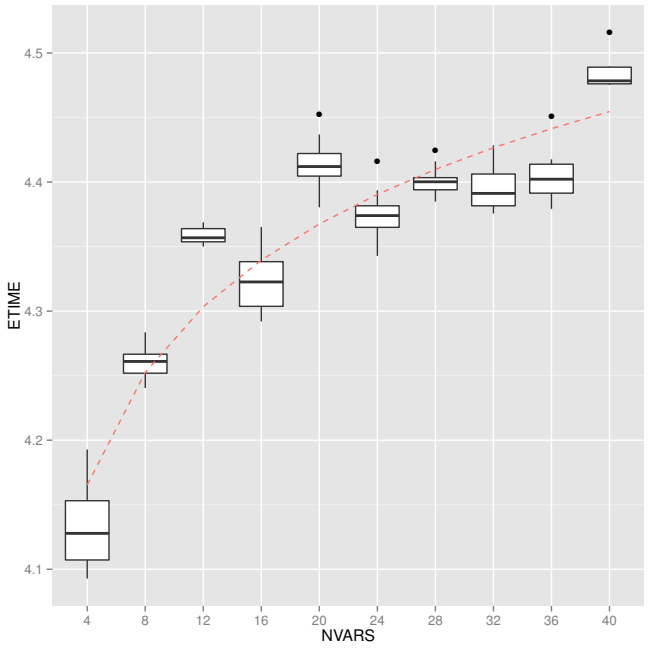

Fig. 7. Execution time taken by the obfuscated program to compute 1,000,000 opaque constants. Opaque constants are based on a k-clique problem defined by 3SAT formula in **NVARS** propositional variables. The red line represents the log model.

The trend of ETIME seems to suggest a logarithmic dependency with NVARS. To validate this observation, we fit experimental data with the subsequent log model:

$$\text{ETIME} = a + b \cdot \log(\text{NVARS})$$

Table II show the result of the model estimated with a linear least squares regression. The first column reports the name of parameter that is estimated (a and b). The second column reports the estimated value for the parameter. Then the third

and fourth columns report, respectively, the standard error and the p-value of the t-test. As we can see, the parameters can be estimated with a very large confidence[4]. The estimated log model is shown in Figure 7 as an interpolating red dashed line.

TABLE II
ESTIMATED COEFFICIENTS OF THE LOGARITHMIC MODEL FOR THE EXECUTION TIME TAKEN TO EVALUATE 1,000,000 OPAQUE CONSTANTS.

Parameter	Estimation	Std. Err.	P-value
a	3.99126	0.04789	4.79e-13
b	0.12560	0.01607	5.17e-05

Based on this experiment, we can answer to research question RQ_2 in this way:

A program obfuscated with opaque constants based on k-clique is affected by a very low runtime overhead. The runtime overhead increases logarithmically with the the number of propositional variables used in the 3SAT formula according to this model:

$$ETIME = 3.99 + 0.12 \cdot \log(NVARS)$$

D. RQ_3: Resilience of Opaque Constants based on 3SAT

In the third experiment, we evaluate the resilience of the approach based on 3SAT proposed by Moser et al.

We estimate the resilience of obfuscation as the amount of time taken by a static analysis tool to break obfuscation. Breaking obfuscation means to guess that for any possible input value, the same opaque value is returned by the obfuscated function. This boils down to detect that some execution paths in the control flow graph are always executed (e.g., `opaque_bit=0` in Figure 4) and some other paths are never executed (e.g., `opaque_bit=1`) because they are unfeasible.

To this aim, we resort to symbolic execution [14]. KLEE [15] is a state-of-the-art symbolic execution tool that automatically generates (several sets of) input values for a C program or a C function. The control flow graph of the program under analysis is visited and path conditions are collected along branch statements. Path conditions are solved to compute concrete input values that would make the execution traverse the visited branches.

With respect to the opaque constant bit based on 3SAT in Figure 4, KLEE will try to explore the branch at line 21 and generate a set input values (values of $v_1 ... v_n$) that makes the execution reach the branch with `opaque_bit=1`. To execute line 21 all the propositional clauses should evaluate to *true*. However, there is no assignment that makes the execution reach that point, because the 3SAT formula is unsatisfiable by construction. To understand this fact, KLEE needs to solve the 3SAT problem.

The experimental procedure consists in measuring how long KLEE takes (i.e., ETIME) to fully analyse a function that computes an opaque constant bit based on 3SAT, similar to the function in Figure 4.

[4]A confidence of 95% requires a p-value< 0.05. In our case, p-values are always much smaller.

We use 3SAT formulas with an increasing number of propositional variables NVARS. For each formula the number of clauses NCLS is set to $\text{NCLS} = \lfloor 4.3 \cdot \text{NVARS} \rfloor$ as suggested by Selman's et al. to maximize the likelihood of having a hard-to-solve SAT problem (see in Section IV-B). For each value of NVARS, we experiment with 10 different unsatisfiable 3SAT formulas. The measurement of the time ETIME taken by KLEE to analyse each 3SAT formula is repeated 25 times.

TABLE III
EXECUTION TIME TAKEN BY KLEE TO ANALYSE A PROGRAM WITH A 1-BIT 3SAT OPAQUE CONSTANT USING NVARS PROP. VARIABLES.

NVARS	ETIME	CI
4	0.24	0.01
8	1.08	0.04
12	6.73	0.65
16	43.63	5.16
20	530.48	143.75

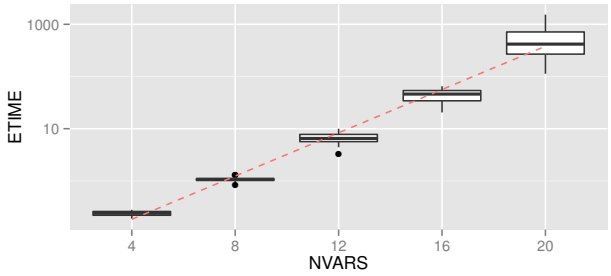

Fig. 8. Execution time (y axis is logarithmic) taken by KLEE to analyse a program with a 1-bit opaque constant based on a 3SAT problem in NVARS propositional variables.

Table III shows the summary of experimental data. The first column reports the number of propositional variables NVAR, the second column reports the the average ETIME of KLEE in seconds and the third column shows the the 95% confidence interval for the average. The same data are also shown in Figure 8 with the y axis in logarithmic scale. As expected, the trend looks exponential (linear on a logarithmic axis), so we try and fit ETIME against the number of propositional variables using an exponential model:

$$\text{ETIME} = e^{a+b \cdot \text{NVARS}}$$

The obtained model is shown in Table IV. The table columns report, for each model parameter, the estimated value, the standard error and the p-value of the t-test.

TABLE IV
ESTIMATION OF THE MODEL TO FIT KLEE EXECUTION TIME DATA PRESENTED IN TABLE III.

Parameter	Estimation	Std. Err.	P-value
a	-3.60377	0.34213	0.001827
b	0.47717	0.02579	0.000345

Based on the empirical evidence, we can answer to RQ_3 in this way:

A state-of-the-art symbolic execution tool can analyse opaque constants based on 3SAT, and the analysis time grows exponentially with the number of propositional variables. Furthermore to analyse a function that compute a 1-bit opaque constant the analysis time can be estimated with this model:

$$\text{ETIME}_{\text{3SAT}} = e^{-3.60+0.48\cdot\text{NVARS}}$$

E. RQ_4: Resilience of Opaque Constants based on k-clique

Similarly to the experiment on opaque constants based on 3SAT, here we study the resilience of opaque constants based on k-clique. Also in this case we use KLEE as state-of-the-art static analysis tool.

However, it turns out that collecting time measures for the k-clique problem is not feasible. According to our approach, the smallest k-clique problem is reduced from the smallest non-trivial 3SAT problem, i.e. with NVARS=4. It corresponds to a 17-clique in a graph with 51 vertexes. When analysing the code for such small problem, KLEE is not able to complete the analysis after having run for more than 9 days.

Thus, we have to resort to a different experiment, with an easier problem instance. The new problem consists in measuring how long KLEE takes to analyse simpler k-clique problem, that is not derived from a (unsatisfiable) 3SAT formula. We use smaller random generated graphs (with less than 51 vertexes) instead of graphs derived from a 3SAT formulas. In this case, graphs are randomly generated. We we can not guarantee that they do not contain a k-clique. Henceforth it is possible that KLEE needs less time to analyse the program obfuscated with on these graph. Thus, we measure a lower-bound of actual ETIME for unsatisfiable problems.

In detail, the experimental procedure consists in generating random graphs of increasing size and checking for the presence of cliques. The number of vertexes NVGR of graphs is set equal to three times the size KCLQ of the clique, namely $\text{NVGR} = 3 \cdot \text{KCLQ}$. For each pair of vertexes, we randomly add an edge to the graph with probability $P = 0.85$. The experiment is conducted with different values of KCLQ that varies between 3 and 8. The analysis of each opaque constant with KLEE is repeated 25 times, and its execution time ETIME is measured.

Table V summaries the empirical data. For each size of the clique KCLQ, the table shows the mean of ETIME in seconds and the 95% confidence interval (CI).

TABLE V
MEAN EXECUTION TIME AND 95% CONFIDENCE INTERVAL FOR THE MEAN, REQUIRED BY KLEE TO ANALYSE A PROGRAM WITH ONE-BIT OPAQUE CONSTANT BASED ON k-CLIQUE.

KCLQ	ETIME	CI
3	1.28	0.12
4	13.54	1.26
5	96.83	16.59
6	569.09	140.62
7	3,722.56	1,040.93
8	14,358.15	2,389.90

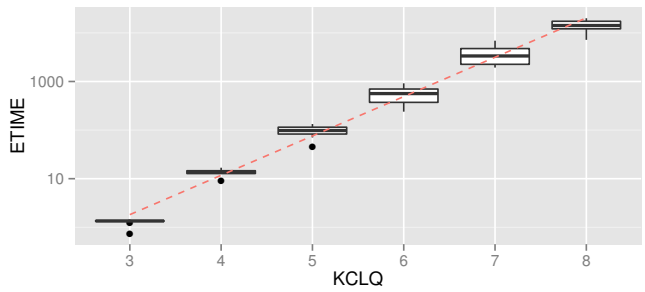

Fig. 9. Execution time (y axis is logarithmic) taken by KLEE to analyse a program with one-bit opaque constant based on random graphs whose size is $\text{NVGR} = 3 \cdot \text{KCLQ}$.

Figure 9 reports the boxplot of execution time for different sizes of the clique (y axis has a logarithmic scale). The graph suggests an exponential trend (linear in the logarithmic scale), so we try and fit ETIME against the number of vertexes in the clique KCLQ using an exponential model:

$$\text{ETIME} = e^{a+b\cdot\text{KCLQ}}$$

Table VI shows the resulting model, for each parameter (a and b) the estimated value is shown (second column) with the standard error and p-value (third and fourth columns respectively). The estimated model is shown in Figure 9 as a red dashed line.

TABLE VI
ESTIMATION OF THE MODEL TO FIT KLEE EXECUTION TIME DATA PRESENTED IN TABLE V.

Parameter	Estimation	Std. Err.	P-value
a	-4.9915	0.4221	0.000293
b	1.8641	0.0733	1.42e-05

Based on these results, we can formulate the subsequent answer to RQ_4:

A state-of-the-art symbolic execution tool can not analyse opaque constants based on k-clique. The analysis on simplified versions of opaque constants based on k-clique takes exponential time in the size of the clique KCLQ. Furthermore, to analyse a function that computes a 1-bit opaque constant, the analysis time can be underestimated with this model:

$$\text{ETIME}_{\text{K}-\text{Clique}} \geq e^{-4.99+1.86\cdot\text{KCLQ}}$$

F. Considerations about Requirements

Here we discuss the results of the empirical investigation with respect to the requirements of opaque constants defined in Section III-B.

Requirement Req_1: Our transformation is sound because we check that the propositional formula used in the transformation is unsatisfiable, so the program semantic is preserved.

Requirement Req_2: We have a direct measurement for the time required to analyse opaque constants based on 3SAT

(see RQ_3). However, we could not directly measure now long KLEE takes to analyse an opaque constant based on k-clique (RQ_4). We could measure the tool execution time only on simplified problems. If we assume that we can extend the validity of results for RQ_4 to a larger problem size, we can speculate on and predict the amount of time needed to complete the analysis. Considering that in our approach we have that $\text{KCLQ} = \lfloor 4.3 \cdot \text{NVARS} \rfloor$, we can rewrite the model fitted on RQ_4 in terms of NVARS:

$$\text{ETIME}_{\text{K-Clique}} \geq e^{-4.99+1.86 \cdot \text{KCLQ}} = e^{-4.99+1.86 \cdot \lfloor 4.3 \cdot \text{NVARS} \rfloor}$$

Using this interpolated model for $\text{ETIME}_{\text{K-Clique}}$, we can predict the time required to analyse a k-clique problem. Let's consider the problem instance obtained from a 3SAT formula of 17 clauses in 4 propositional variables. $\text{ETIME}_{\text{K-Clique}}$ should be about $e^{-4.99+1.86 \cdot 17} \simeq 3.7 \cdot 10^{11}$ seconds, that is more than 11,000 years. Thus, we can claim that our approach satisfies requirement Req_2, because static analysis would take years to attack an opaque constant based k-clique.

Requirement Req_3: Our tool took milliseconds to generate the opaque constant code for the k-clique problem discussed above, i.e. based on 3SAT formula in 4 propositional variable and 17 clauses. So we can claim that requirement Req_3 is also fully met, because the time taken to obfuscate constant values is short and, specifically, much shorter than the time taken to attack them.

Requirement Req_4: The the computation of the value of an opaque constant based on k-clique is quite fast (see RQ_2) and it grows logarithmically with the problem size. Thus, we can claim that requirement Req_4 is also fully met. Thus, we can formulate the subsequent statement:

> *Opaque constants based on the k-clique problem are sound (Req_1); hard to guess with symbolic execution (Req_2); fast to generate with our approach (Req_3); and fast and scalable to compute at execution time (Req_4).*

To complete the discussion, we compare the amount of time required to analyse (i.e. to attack) opaque constants based on 3SAT-based and those based on k-clique. So, we compute a lower bound for the ratio between $\text{ETIME}_{\text{K-Clique}}$ and $\text{ETIME}_{\text{3SAT}}$ as:

$$\frac{\text{ETIME}_{\text{K-Clique}}}{\text{ETIME}_{\text{3SAT}}} \geq \frac{e^{-4.99+1.86 \cdot \lfloor 4.3 \cdot \text{NVARS} \rfloor}}{e^{-3.4+0.48\text{NVARS}}} \geq 0.2 \cdot 10^{3.2 \cdot \text{NVARS}} \quad (5)$$

This result clearly shows for KLEE our opaque constants based on a k-clique problem are (exponentially) more difficult to analyse than opaque constants proposed by Moser et al. based on a 3SAT problem. So, our opaque constants clearly improve the resilience of previous approaches, while they require a similar cost at obfuscation time and at execution time.

VI. Related Work

The most related work by Moser et al. [4] has already extensively described in this paper. We overcome their limitations by requiring static analysis to run for (exponentially) longer time to analyse our opaque constants. Other relevant techniques to tackle the problem of generating opaque constants and opaque predicates were presented by Collberg et al. [5][10][16] and by Wang et al. [17].

The techniques proposed by Collberg et al. leverage the hardness (undecidability, in general [18]) of the statically must/may point-to analysis problem. The opaque predicate is formulated on a dynamic data structure (e.g. a graph) that is difficult to analyse statically, because continuously updated at runtime.

Wang et al. [17] presented a technique to obfuscate predicates that trigger malware behaviours. The technique aims at preventing symbolic execution to devise which conditions satisfy a certain predicate. To this aim, they leverage mathematical conjectures, such as the Collatz's one.

However, the arguments of Collberg et al. and Wang et al. to support the strength of their approaches are informal. Conversely, we adopted an empirical framework to show that our approach defeats a state-of-the-art symbolic execution tool.

Work by cryptographers, such as Barak et al. [19], aims to a formal definition for the concept of obfuscation, proving possibility or impossibility theorems for the existence of different strength class of obfuscations, such as indistinguishability obfuscation [20]. Other work focuses on proposing obfuscation implementations for practitioners and on providing measures or speculations on obfuscation's strength and ability to delay attacks.

Jakubowski et al. [21] used code metrics (size, cyclomatic number and knot count) to measure code complexity as a proxy of human understanding effort.

The term *"resilience"* was proposed by Collberg et al. [16] as a quality related to how difficult is an obfuscated program to be automatically de-obfuscated. Karnick et al. [22] measure resilience as the number of errors generated when decompiling the obfuscated code. Sutherland at al. [23] relied on a program binary instrumentation tool to measure the fraction of the obfuscating transformations that attackers can undo automatically. Udupa et al. [24] evaluated the effectiveness of control flow flattening obfuscation, by measuring how long a combination of static and dynamic analysis takes to perform the automatic de-obfuscation. We also used an automatic tool to asses the resilience of our obfuscation approach.

VII. Conclusions and Future Work

Opaque constants are cornerstone features to extend and improve existing obfuscation transformations. We presented a novel approach for opaque constants that are sound, hard to guess, fast to generate and fast to compute. According to our empirical investigation, symbolic execution does not threaten our opaque constants, because it would require too long analysis time (many years to complete).

As future work, we plan to investigate the effect of our obfuscation on human comprehension. Human participants will be involved in a controlled experiment to measure how hard is to tamper with obfuscated code.

REFERENCES

[1] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, "Guest editors' introduction: Software protection," *Software, IEEE*, vol. 28, no. 2, pp. 24–27, 2011.

[2] G. Wroblewski, "General method of program code obfuscation. phd thesis." Ph.D. dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002., 2002.

[3] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques," *Empirical Software Engineering*, vol. 19, pp. 1040–1074, 2014.

[4] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*. IEEE, 2007, pp. 421–430.

[5] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. of Computer Science, The Univ. of Auckland, Technical Report 148, 1997.

[6] C. Zarate, S. L. Garfinkel, A. Heffernan, K. Gorak, and S. Horras, "A survey of xor as a digital obfuscation technique in a corpus of real data," DTIC Document, Tech. Rep., 2014.

[7] J. Cannell, "Obfuscation: Malwares best friend," March 2013. [Online]. Available: http://blog.malwarebytes.org/intelligence/2013/03/obfuscation-malwares-best-friend/

[8] W. Zhu and C. Thomborson, "A provable scheme for homomorphic obfuscation in software security," in *The IASTED International Conference on Communication, Network and Information Security, CNIS*, vol. 5, 2005, pp. 208–212.

[9] H. L. Garner, "The residue number system," *Electronic Computers, IRE Transactions on*, vol. EC-8, no. 2, pp. 140–147, June 1959.

[10] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1998, pp. 184–196.

[11] B. Selman, D. G. Mitchell, and H. J. Levesque, "Generating hard satisfiability problems," *Artificial intelligence*, vol. 81, no. 1, pp. 17–29, 1996.

[12] M. R. Garey and D. S. Johnson, *Computers and intractability*. wh freeman New York, 2002, vol. 29.

[13] R. M. Karp, *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Boston, MA: Springer US, 1972, ch. Reducibility among Combinatorial Problems, pp. 85–103. [Online]. Available: http://dx.doi.org/10.1007/978-1-4684-2001-2_9

[14] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[15] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[16] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation: tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 735–746, August 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=636196.636198

[17] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *Computer Security–ESORICS 2011*. Springer, 2011, pp. 210–226.

[18] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[19] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," *Journal of the ACM (JACM)*, vol. 59, no. 2, p. 6, 2012.

[20] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*. IEEE, 2013, pp. 40–49.

[21] M. H. Jakubowski, C. W. Saw, and R. Venkatesan, "Iterated transformations and quantitative metrics for software protection." in *SECRYPT*, 2009, pp. 359–368.

[22] M. Karnick, J. MacBride, S. McGinnis, Y. Tang, and R. Ramachandran, "A qualitative analysis of java obfuscation," in *Proceedings of 10th IASTED International Conference on Software Engineering and Applications, Dallas TX, USA*, 2006.

[23] I. Sutherland, G. E. Kalb, A. Blyth, and G. Mulley, "An empirical examination of the reverse engineering process for binary files," *Computers & Security*, vol. 25, no. 3, pp. 221–228, 2006.

[24] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *Proceedings of the 12th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 45–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=1107841.1108171