The Journal of Systems and Software 162 (2020) 110492

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Experimental assessment of XOR-Masking data obfuscation based on K-Clique opaque constants

Roberto Fellin[a], Mariano Ceccato[b,c,*]

[a] University of Trento, Via Sommarive 9, 38123, Trento, Italy
[b] University of Verona: Strada le Grazie 15, 37134 Verona, Italy
[c] Fondazione Bruno Kessler: Via Sommarive 18, 38123 Trento, Itlay

ARTICLE INFO

Article history:
Received 19 March 2019
Revised 24 October 2019
Accepted 4 December 2019
Available online 9 December 2019

Keywords:
Data obfuscation
Program transformation
Obfuscation overhead

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. In XOR-Masking, a constant mask is used to obfuscate data, but this mask must be hidden too, in order to keep the obfuscation resilient to attacks.

In this paper, we present a novel extension of XOR-Masking where the mask is an opaque constant, i.e. a value that is difficult to guess by static analysis. In fact, opaque constants are constructed such that static analysis should solve the *k*-clique problem, which is known to be NP-complete, to identify the mask value.

In our experimental assessment we apply obfuscation to 12 real Java applications. We observe that obfuscation does not alter the program correctness and we record performance overhead due to obfuscation, in terms of execution time and memory consumption.

1. 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 Falcarin et al., 2011). 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 (Wroblewski, 2002). 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.

XOR-Masking is a popular data obfuscation scheme that uses a *mask* to encode sensitive program data. However, in case the attacker is able to identify the mask value, sensitive data can be easily decoded statically, and leak to the attacker in clear. So, the mask should not be easy to detect with malicious reverse engineering, for instance the mask should not be a static constant.

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. In a companion paper (Tiella and Ceccato, 2017) we presented an approach to automatically obtain opaque constants based on the *k*-clique problem, that overcomes the limitations of previous art (Moser et al., 2007) and that seems promising for practical adoption.

In our previous paper, we also empirically shown that an opaque constant is fast to generate at obfuscation time, but it is extremely hard to attack using static analysis. In this paper, we build on top of our previous results and we use our opaque constants to implement a novel data obfuscation scheme. We extend XOR-Masking data obfuscation, by using an opaque constant as mask, such that its resilience to static analysis dramatically improves. Moreover, while in the former paper, we just used synthetic examples and way too simple programs, in this paper we apply data obfuscation on real code.

We demonstrate that data obfuscation is sound, because it does not alter programs by introduction implementation errors, and we measure the practical cost of data obfuscation on realistic

* Corresponding author.

https://doi.org/10.1016/j.jss.2019.110492
0164-1212/© 2019 Elsevier Inc. All rights reserved.

case studies, by measure execution time and memory consumption overhead.

The paper is structured as follows. The problem of strengthening data obfuscation in exposed in Section 2. The background of opaque constants with *k*-clique is covered in Section 3. Then, our approach to use opaque constants to define a novel XOR-Masking data obfuscation scheme is presented in Section 4, which is empirically evaluated in Section 5. Section 6 comments related work and, lastly, Section 7 summaries conclusions and future work.

2. Problem definition

Obfuscation (Collberg et al., 1997) 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* is a program transformation approach meant to complicate the representation of data to make them harder to comprehend.

XOR Masking is a quite frequently (Zarate et al., 2014) used data obfuscation transformation based on the bitwise XOR operator. Variants of XOR-Masking are known to be employed by practitioners and by malware (Cannell, 2013). This obfuscation consists in encoding a clear value by computing its bitwise XOR (denoted with ^) 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.

Fig. 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 their clear values are needed. At line 3, *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, then, decoded in the last line, before it is printed.

Data encoded with XOR-Masking are vulnerable to attacks based on static analysis. In fact, the mask *p* used in XOR-Masking is a static constant that, once extracted from the code, can be used to decode and obtain the clear value of obfuscated variables. A way to make this obfuscation technique more resistant to static analysis attacks, is to turn this constant into a different software entity that is harder to identify statically. An option is to replace the mask value with an *opaque constant* (Collberg et al., 1998). 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.

2.1. Attack model

Our attack model is based on the outcome of series of studies conducted with professional hacker teams and with a public challenge, to observe how protected programs are attacked (Ceccato et al., 2017; 2019). The facts that could be observed from these studies include the following:

- When tampering with a program, dynamic analysis seems a natural choice. In fact, an attacker might set breakpoints to suspend the execution at interesting points in time, to read the memory and, possibly, to alter values;
- In case code-protections are deployed to mitigate attacks, hackers revise their strategy to choose the path of minimal resistance and minimize the attack effort. E.g., in case *anti-debugging* (Abrath et al., 2016) is deployed to prevent debugging, instead of defeating this protection and be able to attach a debugger, attackers prefer to switch to static analysis;
- Static analysis is one of the main approaches (together with dynamic analysis) used by hackers to attack protected code;
- Each protection approach has strong and weak points, weak points could be exploited to defeat or work around a protection. Thus, different code protection approaches should be integrated and deployed together, not only to protect the code from tampering attacks, but also to protect each other protection from attackers' attempts to defeat them.

To identify and extract sensitive data from a program, dynamic analysis seems a natural choice. In fact, an attacker might set breakpoints to suspend execution and read sensitive data in memory at run-time. To protect sensitive data, the first line of defense consists in blocking dynamic analysis, for instance by preventing a debugger to attach using anti-debugging. When dynamic analysis is too expensive or too time-consuming, attacker shown to revise their strategy. Instead of elaborating a very complex solution and continue with dynamic analysis, static analysis becomes a valuable alternative. Thus, the second line of defense consists in contrasting static analysis, to prevent that attackers can easily identify and extract sensitive data from the static binary.

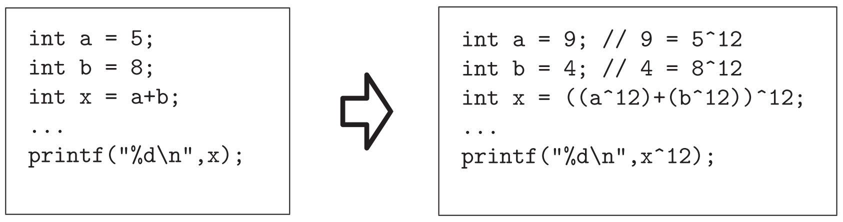
The contribution of this paper is positioned in the scope of this second line of defense, to prevent attacks based on static analysis. In other words, we consider dynamic analysis out of the scope, for instance because it is not applicable due to anti-debugging.

However, to be effective in practice, protection approaches against static analysis and against dynamic analysis should be paired and deployed together, to defend each other weak point and, together, to defend the code from malicious reverse engineering.

Similarly to what assumed by related work (Moser et al., 2007; Collberg et al., 1998), in this paper we also assume that an 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.

2.2. Requirements of data obfuscation with opaque constants

The first requirement descends directly by Collberg's definition of obfuscation transformation (Collberg et al., 1997): the obfuscated program must behaves as the original clear program on valid inputs. This means that, the end-user should notice no dissimilarity between the clear and the obfuscated program, such as crashes or different results, across *all* the implemented features, i.e.:



```
int a = 5;
int b = 8;
int x = a+b;
...
printf("%d\n",x);
```

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

Fig. 1. Example of XOR-Masking.

Requirement Req $_1$: *The obfuscation transformation must be sound.*

The second requirement for data obfuscation is that it should be resilient and difficult to break. *Resilience* is defined as a measure of how well a transformation holds up under attack by an automatic deobfuscator (Collberg et al., 1998). Attackers in fact might adopt tools to perform malicious reverse engineering and try to extract decoded values from a program. For example, if an opaque value is computed starting from constant values, it could be easily recovered using static analysis techniques such as constant propagation. To avoid this analysis, an opaque value should be computed starting from random values or program inputs. Moreover, recovering an opaque constant should require the attacker to solve a known hard problem, i.e. an NP-hard problem. So our second requirement is the following:

Requirement Req $_2$: *Guessing the mask values to decode obfuscated values 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 $_3$: *Obfuscation should be practically fast to apply.*

Additional code is inserted to a program, to compute opaque constants at execution time. When this additional code is complex, or it is executed very often, 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, a clear data should be decoded in polynomial time. Thus, the last requirement is:

Requirement Req $_4$: *Performance overhead of obfuscated programs should be acceptable.*

2.3. Considerations about requirements

As we will discuss in the next sections, in this paper we propose an extension of XOR-Masking data obfuscation, where an opaque constant is used to protect the mask from static analysis. Opaque constants are based on the *k*-clique approach, proposed by us in a previous work Tiella and Ceccato (2017). In that paper, we also validated some properties of these opaque constants, so that some of the previous requirements are satisfied, but some others are still open.

- ✗ **Req**$_1$: Even if an obfuscation scheme is theoretically sound, we could have overlooked important details or have committed implementation mistake. Thus, we need to check if program behaviour is altered by obfuscation, to guarantee that no defect has been introduced by our transformation;
- √ **Req**$_2$: Breaking data obfuscation is hard, because guessing an opaque constant based on *k*-clique with static analysis is hard. In fact, we already shown Tiella and Ceccato (2017) that a state-of-the-art symbolic execution tool could not complete the analysis of the piece of obfuscated code that computes opaque constants at runtime. Moreover, a simplified version of this code required exponential analysis time (more than 11,000 years for normal sized problems). Analysis time is estimated to take exponentially longer than a similar previous opaque constant approach, i.e. based on 3SAT by Moser et al. (2007);
- √ **Req**$_3$: An obfuscation is practically fast, because constructing the opaque constant at obfuscation time is not a hard problem. In fact, the obfuscation time is dominated by the time needed to generate a 3SAT unsatisfiable formula. Our past experimental assessment Tiella and Ceccato (2017) showed that it takes less than one second to generate a very hard 3SAT formula, with up to 200 variables;
- ✗ **Req**$_4$: Computing the opaque constant at execution time is expected to be quite fast. This requirement is only partially meet by our previous experimental validation. In fact, in our previous paper, we just shown that computing the opaque constant is fast if done once, but an obfuscated program might require to compute it multiple times. We still need to investigate the performance overhead of a *real* obfuscated program, where obfuscated data are used many times. This second dimension has not been investigated so far.

In the next sections, we will present our approach based on *k*-clique, to craft resilient and manageable opaque constants. We will also recall how our solution addresses our requirements.

3. Opaque constant based on K-Clique

Our opaque constants are based on the *k*-clique problem, a problem known to be NP-hard Garey and Johnson (2002). This approach allows to craft resilient opaque constants that 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.

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.

3.1. Generation of a hard 3SAT problem

Our approach starts from an unsatisfiable propositional formula for 3SAT problem. 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] $\{v_1, v_2, \ldots, v_n\}$ and with k 3-literals

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

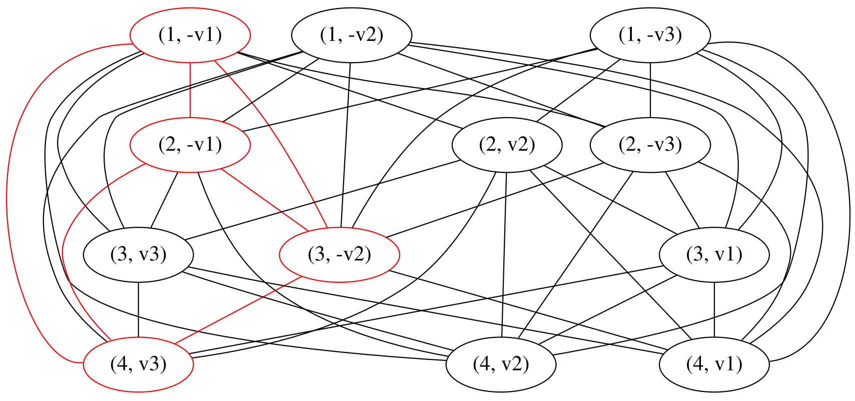


Fig. 2. Graph obtained by means of the Karp's reduction of a 3SAT problem to a k-clique problem for the formula in Example 2.

clauses:

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

The 3SAT problem consists in identifying a truth assignment for variables $v_1, v_2, \ldots, v_n$ 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:

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

To generate a hard instance of the 3SAT problem, we adopt the guidelines proposed by Selman et al. (1996) 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. In a past work Tiella and Ceccato (2017), we assessed this probability and the time required to check the propositional formula to be unsat. The total time required to generate a 3SAT unsatisfiable formula in up to 200 variables was less than one second with a very high probability. This allows us to conclude that requirement $\mathbf{Req}_3$ is met, because the obfuscating tool can assess the propositional formula fast enough.

Then, this 3SAT unsatisfiable formula is turned into a k-clique problem, as described in the following.

3.2. 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 Eq. (1). We construct a graph $G_\varphi = (V, E)$, whose vertexes V and edges E are defined according, respectively, to Eq. (3) and Eq. (4).

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

$$E = \{((i_1, \alpha_{i_1,j_1}), (i_2, \alpha_{i_2,j_2}))\ | i_1 \neq i_2 \text{ and } \alpha_{i_1,j_1} \wedge \alpha_{i_2,j_2} \text{ satisfiable}\} \quad (4)$$

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.

Fig. 2 shows the graph corresponding to the example propositional formula in Eq. (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).

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 $\bar{V} = \{(i, \alpha_{i,j_i}), i = 1, \ldots, k\}$ 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.

In our running example, the propositional formula in Eq. (2) has 4 clauses and it is satisfiable. So the graph in Fig. 2 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 Eq. (3)), starting from a 3SAT formula φ with k clauses, the resulting graph G_φ will have $3k$ nodes.

3.3. 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

[2] Here we intend the reduction as proposed by Karp (1972). Informally, a *reduction* is the transformation of a decision problem into another by means of an algorithm that executes in polynomial time.

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

to be turned into an opaque constant is split into n_b-bits, $c = b_0 b_1 \dots b_{n_b - 1}$. 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.

Fig. 3 shows a snippet of code that generates an opaque bit, i.e. a single bit of an opaque constant c. The function call at Line randomly generates a set of vertexes. The `generate_subset(idx,n,s)` randomly assigns elements from the set $\{0, 1, 2, \dots, s-1\}$ 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 and Line 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). The chunk of code is repeated for all n_b.

We already measured Tiella and Ceccato (2017) how long the code in Fig. 3 takes to compute the opaque at runtime by an obfuscated program. The result shown the 16-bits opaque constants based on k-clique are computed quite fast, they take between 4 and 4.5 seconds for computing 1 million opaque constants. Moreover, computation time increases logarithmically with an increasing number of propositional variables used in the 3SAT formula. This means that requirement $\mathbf{Req}_4$ is at least partially met, because computing an opaque value at runtime is fast. However, a more complete empirical assessment is required, to measure the actual slowdown of real-world programs when obfuscated with this approach.

4. Xor-Masking data obfuscation based on opaque constants

Here we present our approach based on opaque constants to propose a novel obfuscation schema for XOR-Masking. We implemented three different variants of data obfuscation, they are:

- XOR-Masking with constant mask;
- XOR-Masking with opaque mask; and
- XOR-Masking with cached opaque mask.

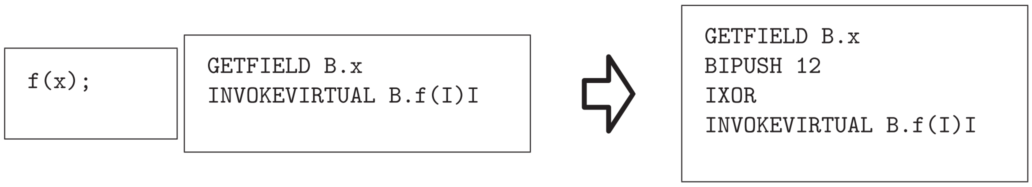
4.1. XOR-Masking with constant mask

This obfuscation is shown in Fig. 1. The obfuscation transformation has been implemented as a Java bytecode transformation in ASM[3]. Applying the transformation at bytecode level makes it compatible with all the languages that compile to Java bytecode, including not only Java, but also JRuby, Jython, Scala and Kotlin. Additionally, the obfuscation is done directly on the bytecode, so the source code is not needed and we can obfuscate also libraries.

Obfuscation applies to class fields of types *int, long, float, double, char* and *string*. For *char, int* and *float* types a 32-bit mask is used, while for *long* and *double* types a 64-bit mask is used. Variables of type *string* are obfuscated by iterating over all the characters and applying a 32-bit mask to each character.

Field uses: Fig. 4 shows how XOR-Masking transforms field read accesses, i.e. field uses. The code in the left-hand side shows a

[3] https://asm.ow2.io/

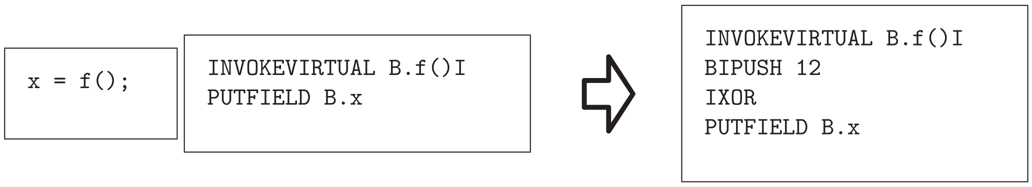


```
f(x);
```

```
GETFIELD B.x
INVOKEVIRTUAL B.f(I)I
```

```
GETFIELD B.x
BIPUSH 12
IXOR
INVOKEVIRTUAL B.f(I)I
```

Fig. 4. Obfuscation of filed uses (Java code, compiled clear bytecode and obfuscated bytecode).

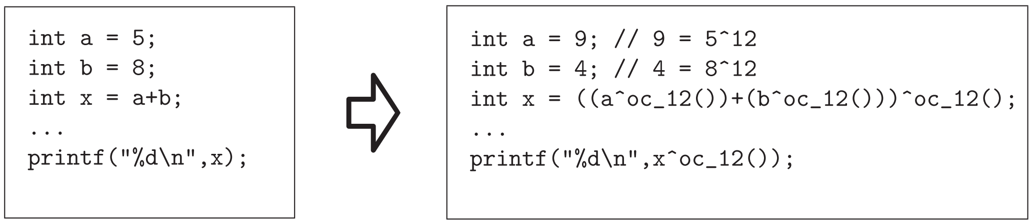


```
x = f();
```

```
INVOKEVIRTUAL B.f()I
PUTFIELD B.x
```

```
INVOKEVIRTUAL B.f()I
BIPUSH 12
IXOR
PUTFIELD B.x
```

Fig. 5. Obfuscation of filed assignments (Java code, compiled clear bytecode and obfuscated bytecode).



```
int a = 5;
int b = 8;
int x = a+b;
...
printf("%d\n",x);
```

```
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. 6. Example of XOR-Masking with opaque mask.

piece of Java that accesses the field x and uses its value as actual parameter to call the method f. This code compiles to the bytecode shown in the center, the field is read by GETFIELD opcode and stored in the operand stack. The subsequent opcode INVOKEVIRTUAL calls the method f that expects parameters to be available in the operand stack.

Our transformation applies after the field is read, by adding two brand new opcodes. The obfucated code is shown in Fig. 4, right-hand side. BIPUSH pushes the constant mask (value 12) on the stack, next to the field value. Then, IXOR consumes two entries in the stack (the filed value and the mask value), it computes the bitwise xor and pushes the results into the stack. This corresponds to decoding the obfuscated values of field x, needed by the subsequent opcode INVOKEVIRTUAL, that calls method f with the clear value of x as actual parameter.

Field defs: Fig. 5 shows the second pattern, in case of assigning a value to a field that should be obfuscated. In the left-hand side, a piece of Java code contains an assignment to x. The corresponding compiled bytecode is shown in the center. The method call (opcode INVOKEVIRTUAL) stores a return value in the operand stack, and the opcode PUTFIELD consumes it and assigns it to x. Our transformation adds two brand new opcodes (see right-hand side). After method f returned, the clear return value in the stack is encoded by pushing the mask in the stack (BIPUSH opcode) and by computing the bitwise xor (IXOR opcode). The encoded value, now available in the stack, is then assigned to the field.

Slightingly similar obfuscation patterns apply in case the filed to obfuscate is a *static* filed, because the patterns have to match opcodes GETSTATIC and PUTSTATIC instead.

4.2. XOR-masking with opaque mask

The resilience of the previous scheme can be improved by replacing the constant mask with an opaque constant, as shown in Fig. 6. Resilience is improved, because such an opaque mask is harder to identify with static analysis. In fact, static analysis would require to solve an NP-hard problem to figure out the mask.

Instead of using a constant mask (12 in the previous scheme), we call the function (i.e., to oc_12) generated by our approach, that dynamically computes the value of the mask to use in the XOR operations. For a 32-bit mask, this function contains 32 copies of the code in Fig. 3. As a further improve (not shown in the example) the body of the function that computes the opaque mask is inlined in the calling context, so that it is even harder to understand.

In case an opaque constant should be used instead of a constant mask, the patterns in Fig. 4 and Fig. 5 need to be adapted. Before computing the bitwise xor in encode/decode operations, the code of Fig. 3 should be used to dynamically compute the mask value.

4.3. XOR-masking with cached opaque mask

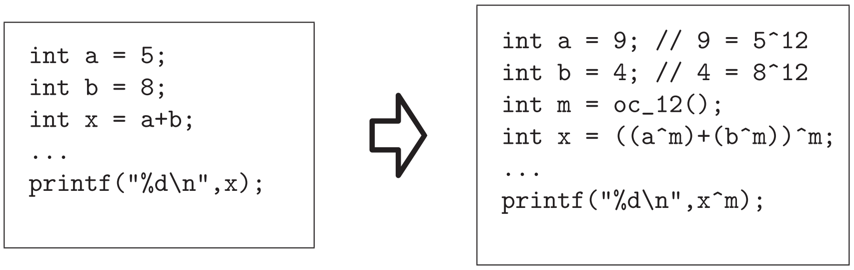
Discarding the mask value after each use and recomputing it each time it is needed is quite secure, because it limits the opportunity for an attacker to detect the mask value. However, this strategy might bring consistent runtime performance overhead, caused by recomputing the mask value too many times. A trade off between obscurity and performance is represented by a third approach, that relies on a cache to still use on opaque constant as mask, but save its value for future uses after it is computed at the first usage.

This obfuscation scheme is shown in Fig. 7, the opaque constant is computed only once the first time it is needed to decode a and b at line 3, by calling the function oc_12. The mask value is stored in the local variable m and used in the subsequent statements.

Also in this case, the body of the function that compute the opaque mask is inlined in the calling context.

5. Empirical evaluation

We performed an empirical evaluation of the proposed approach, to validate the correctness of obfuscated code and the execution overhead caused by data obfuscation.



```
int a = 5;
int b = 8;
int x = a+b;
...
printf("%d\n",x);
```

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

Fig. 7. Example of XOR-Masking with cached opaque mask.

5.1. *Research questions and variable selection*

We formalized our evaluation goals in the following research questions:

- **RQ$_1$:** Does data obfuscation preserve program semantics?
- **RQ$_2$:** What is the practical impact of data obfuscation on performance overhead?
- **RQ$_3$:** What is the impact on performance overhead when increasing the complexity of the *k*-clique problem?

Obfuscation changes a program to make it harder to understand and to analyze, but it should not introduce deviations with respect to the behaviour. The first research question aims at assessing whether the input-output behavior of (real world) programs is preserved after applying obfuscation. However, obfuscation comes at some cost, at least in terms of performance overhead. The latter two research questions are meant to quantify overhead in terms of longer execution time. In particular, the second research question compare execution time of clear and obfuscated programs. The last research question investigates overhead with harder and harder obfuscation schemes.

To answer these research questions, we will consider these metrics:

- **ETIME**: Execution time (in seconds) taken for running a scenario by an (either clear or obfuscated) program, as reported by Maven Surefire Report.
- **MEM**: Amount of memory used by a program, expressed as the MB reported Maven Surefire Report.
- **NVARS**: Number of propositional variables in a 3SAT formula used to create the *k*-clique problem for data obfuscation;
- **TESTS**: Percentage of passing test cases. This value is computed as the ration between the test cases that pass on the obfuscated code and the number of test cases that pass on the original clear code.

5.2. *Case studies*

In this empirical validation, we involve 12 open source Java projects from different domains. Table 1 lists the case studies, together with some statistics. For each case study (first column), the table reports the size in terms of number of classes (second column) lines of code (third column). The fourth and fifth columns report the amount of test code coming with these projects, respectively, as the number of test classes and the number of lines of code. In total, case studies contain more than 390K LoCs and mode than 360K LoCs of test cases.

The case studies considered are:

- Java2word[4]: Java library able to generate Microsoft Word Documents from Java code without any special component.

[4] https://github.com/leonardoanalista/java2word

Table 1
Case studies.

Application	Classes	LoC	Test classes	LoC Test
Java2word	47	4 147	20	2 343
Jfreechart	629	6 698	366	76 856
Xml-Security	463	11 896	197	54 684
Jimfs	58	12 838	50	13 548
Joda-Time	166	70 756	158	72 770
Stream-lib	48	7 925	29	5 429
Gson	74	14 894	101	18 761
cglib	129	14 640	72	7 031
Quartz	204	57 970	69	10 617
LibRec	186	36 921	82	7 107
jackson	418	107 570	498	71 960
GraphHopper	288	44 836	138	22 792
Total	2 710	391 091	1 780	363 898

- Jfreechart[5]: Java chart library which allows to display professional quality charts in Java applications.
- Xml-Security[6]: Java library with the aim of providing implementation of the primary security standards for XML such as Digital Signature and Encryption.
- Jimfs[7]: Java library which implements an in-memory file system for Java 7 and above, implementing the java.nio.file abstract file system APIs.
- Joda-Time[8]: Java library which provides a quality replacement for the Java date and time classes. It is the standard date and time library for Java prior to Java SE 8.
- Stream-lib[9]: Java library for summarizing data in streams for which it is infeasible to store all events.
- Gson[10]: Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object.
- cglib[11]: Java library which provides high level API to generate and transform JAVA byte code.
- Quartz[12]: job scheduling library that can be integrated within virtually any Java application.
- LibRec[13]: Java library for recommender systems (Java version 1.7 or higher required). It implements a suit of state-of-the-art recommendation algorithms, aiming to resolve two classic recommendation tasks: rating prediction and item ranking.
- jackson-databind[14]: Java library with the general-purpose of data-binding functionality and tree-model. It performs JSON

[5] http://www.jfree.org/jfreechart/
[6] http://santuario.apache.org/
[7] https://github.com/google/jimfs
[8] http://www.joda.org/joda-time/
[9] https://github.com/addthis/stream-lib
[10] https://github.com/google/gson
[11] https://github.com/cglib/cglib
[12] https://github.com/quartz-scheduler/quartz
[13] https://github.com/guoguibing/librec
[14] https://github.com/FasterXML/jackson-databind

data-binding and other data formats, as long as parser and generator implementations exist. It builds on Streaming API (stream parser/generator) package, and uses Jackson Annotations for configuration.
- GraphHopper Routing Engine[15]: fast and memory efficient Java routing engine, released under Apache License 2.0. By default it uses OpenStreetMap and GTFS data, but it can import other data sources.

5.3. Obfuscation configuration

To assess obfuscation correctness, we rely on the test cases that come with case studies to test the obfuscated code, and check if test cases still pass after applying data obfuscation. However, to be able and accurately assess the correctness of the obfuscated code, we mean to obfuscate only those fragment of code that we can test to a large extent.

Moreover, the transformed code should not be just trivial methods (e.g., setter and getter methods). To assess the correctness of our obfuscation in realistic and general settings, we mean to obfuscate code with a non-trivial level of complexity. Thus, we adopt the following procedure to select what piece of data to obfuscate:

- *Data access*: Identify what data fields (of primitive types) are accessed by the code of program methods.
- *High test coverage*: Identify those program methods whose test coverage is high, i.e. at least 70% of program code is tested by test cases;
- *Not trivial code*: Program methods to be obfuscated are complex and large, i.e. the number of branches is large.

We need to know what fields are accessed by program methods. However, considering that we will run the program based on the scenarios realized by test cases, instead of opting for static data-access information, we preferred to consider dynamic analysis and count the actual data accesses occurring when test cases are executed.

To this aim, we instrument the code to log when program method code accesses a class field of primitive type (numeric, character or string). Instrumentation has been implemented as a rewrite rule in ASM, that add a trace statement after opcodes that reads/writes class fields of these types. Logs are in the form *method name* $\rightarrow$ *field name*. Then, test cases are executed and execution traces are collected with the code/data mapping.

Test coverage (i.e., what portion of program code is tested by which test cases) is collected using *EclEmma*. EclEmma collects coverage data in terms of the number of branches in each program method that are taken after the whole test suite has been executed. It also reports the static size of each program method, in terms of the total number of branches. We use the total number of branches in a program method to represent method complexity and method size.

Based on these information, we make the final decision of what piece of data to obfuscate. We keep only program methods that access data of primitive types (that, thus, our obfuscation scheme supports) and whose code is tested very well by test cases (coverage $> 70\%$). We discard all the other program methods that do not meet these conditions. The remaining program methods are sorted by according to the number of branches that they contain, and the 10 program methods with the largest number of branches are selected. A data filed is picked for each of these 10 program methods, and marked as target for data obfuscation.

With this procedure, we make sure that we obfuscate a piece of data that is used by non-trivial code that is well tested.

[15] https://github.com/graphhopper/graphhopper

Table 2
Test cases used to assess the correctness of obfuscated code.

Application	Tests
Java2word	154
Jfreechart	2 256
Xml-Security	1 041
Jimfs	5 830
Joda-Time	4 207
Stream-lib	152
Gson	1 050
cglib	302
Quartz	302
LibRec	362
jackson	2 130
GraphHopper	1 375
Total	19 161

The masks are randomly generated, using the default random number generator available in Java. They are either 32-bit or 64-bit values, depending on the length of the type of the field to obfuscate.

When using opaque variants of masks, an instance of k-clique problem is needed. We create it starting from a random 3SAT formula, as explained in Section 3. We set the number of propositional variables to 50 ($\text{NVARS} = 50$) and, consequently, the size k of the k-clique problem is 215 (in fact, $\text{k} = \lfloor 4.3 \cdot \text{NVARS} \rfloor$).

5.4. RQ$_1$:correctness of obfuscated code

Obfuscation consists in a set of transformations to alter the code and make it harder to understand and analyze. However, the obfuscation should be sound, i.e. obfuscation should not change the input-output behavior of a program, for instance by introducing errors in the transformed code. We use test cases to assess the absence of defects in transformed code on a set of programs.

All the case study application are subject to the analysis described above to decide which fields to obfuscate. Three obfuscated versions are produced for each case study, by applying the three variants of data obfuscation, they are XOR-Masking with constant constant mask, with opaque mask and with cached opaque mask. Then, the test cases that come with each case study are run either on the clear program and on three obfuscated versions of it. We record what tests pass on obfuscated code among those that pass on clear code.

The number of test cases is shown in Table 2, each case study in a different line. As we can see in the table, approximately 20k distinct tests have been used to assess the correctness of obfuscated code. 100% of the test cases that pass on the clear programs also pass on the obfuscated programs with all the three transformation variants. Thus, we can answer RQ$_1$ in the following way:

XOR-Masking data obfuscation based on opaque constants with k-clique is sound, because test cases could detect no defect when run on the obfuscated version of realistic programs.

5.5. RQ$_2$: Impact of data obfuscation on runtime overhead

The time taken to execute test cases (ETIME) and the amount of memory used (MEM) by clear and obfuscated code are reported on Maven Surefire Report. Time measurement is repeated 100 times, to improve accuracy and control random errors.

Table 3 compares the execution time of the whole test suite between clear code and obfuscated code, with all the three vari-

Table 3
Execution time (ETIME) overhead caused by XOR-Masking obfuscation.

Application	Clear	Constant	Opaque	Opaque cached
Java2word	0.30	0.30 (+0%)	0.91 (+203%)	0.36 (+10%)
Jfreechart	5.27	5.26 (+0%)	12.05 (+128%)	5.39 (+2%)
Xml-Security	34.41	34.39 (+0%)	79.22 (+130%)	34.18 (+0%)
Jimfs	2.44	2.44 (+0%)	16.34 (+567%)	4.37 (+79%)
Joda-Time	3.58	3.85 (+7%)	5.79 (+61%)	3.90 (+9%)
Stream-lib	119.24	119.17 (+0%)	362.93 (+204%)	119.81 (+0%)
Gson	0.76	0.76 (+0%)	22.58 (+2871%)	0.96 (+26%)
cglib	3.90	3.95 (+1%)	15.99 (+310%)	4.14 (+6%)
Quartz	403.39	403.20(+0%)	456.94 (+13%)	407.78 (+1%)
LibRec	193.52	193.30 (+0%)	269.97 (+39%)	193.72 (+0%)
jackson	7.99	7.99 (+0%)	30.90 (+287%)	8.47 (+6%)
GraphHopper	13.35	13.30 (+0%)	65.53 (+391%)	14.37 (+8%)

ants of XOR-Masking. XOR-Masking with constant mask brings almost unnoticeable slowdown, in fact it is 0% increase in most of the cases, with the only exception of *cglib* where increase is +1% and *Joda-Time* where time increases by 7%.

As expected, when using opaque masks, the execution time increase is larger. The lowest increase is +13% for *Quartz* and the largest is +2,871% for *Gson*. The average increase for this variant of code obfuscation is +433%, which means that the obfuscated code takes between 4 and 5 times longer to execute than the clear code, when an opaque mask is used.

The opaque mask with cache (last column of Table 3) represents tradeoff with intermediate performance. In fact, the overhead is more similar to the case of constant mask (+12% on average), while the mask is still hidden to static analysis, because it is computed at runtime.

Table 4 shows the memory increase due to obfuscation. While XOR-Masking with constant mask does not require additional memory in most of the cases, the other variants record noticeable memory overhead. The variant with opaque mask requires on average more memory than the case when the mask is cached.

The execution time overhead and the memory overhead have a large distribution, especially for the XOR-Masking with opaque constants. To investigate the possible reasons of this large variability across different case studies, we computed the correlations of performance overhead with the impact of obfuscation transformation. In particular we consider the following metrics:

- *Obfuscated fields*: The number of fields that have been obfuscated with XOR-Masking;
- *Static refs*: The number of static references (definitions and uses) to these fields that was found in the code, that required to be obfuscated with XOR-Masking;
- *Dynamic refs*: The number of dynamic references (definition and uses) that have been executed when running the test cases.

General linear model (GLM) incorporates a number of different statistical models: ANOVA, ANCOVA, MANOVA, MANCOVA, ordinary linear regression, *t*-test and *F*-test. We used a general linear model to test the presence of significant correlation between measured overhead (performance difference between obfuscated and clear programs) and the three metrics reported above (number of fields, static refs, dynamic refs). This consists of fitting a model of the *dependent* output variables (time and memory overhead) as a function of the *independent* input variables (number of fields, static refs, dynamic refs). A general linear model allows to test the statistical significance of the influence of all factors on the overhead. We assume significance at 95% confidence level (α=0.05), so we reject the null-hypotheses having p-value < 0.05.

Results of the test with GLM is shown in Table 5. The table reports the p-values of GLM, with statistically significant cases in boldface, each obfuscation variant in a different line. The left-hand side reports the values for the time overhead. The number of obfuscated fields is not correlated with the execution time overhead, in fact we configured the case studies to have a similar number of obfuscated fields (see Section 5.3). The observed slowdown is significantly correlated with the number of dynamic references to obfuscated variables. In fact, the time spent in decoding/encoding values is proportional to the number decoding/encoding operations occurring at run-time. However, significance is observed only for XOR-Masking with *constant* mask and with *opaque* mask. In fact, when the mask is *cached*, only the first encode/decode operation takes time, the subsequent operations can reuse the available mask value. Indeed, when the mask is cached, the observed execution time overhead in Table 3 was quite limited. Execution time overhead is never correlated with the number of static definitions.

The right-hand side of Table 5 reports the correlation results for the memory overhead. In no obfuscation variant we observe significant correlation either with memory overhead. Probably, the lack of correlation is due to the fact that memory overhead has a smaller variation across case studies and it is in general quite low.

Considering these results, we can answer RQ$_2$ in the following way:

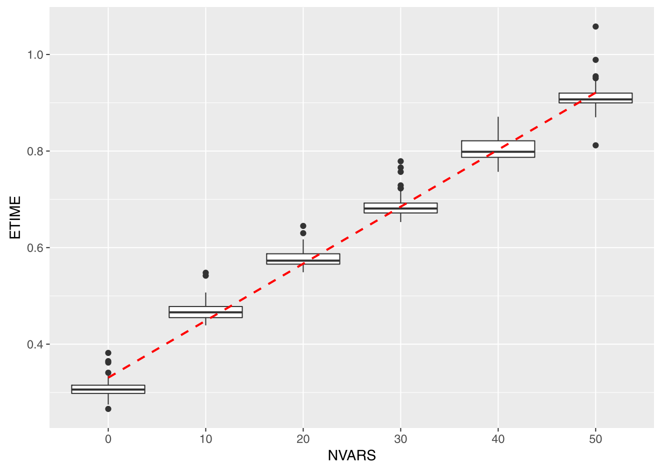
The practical impact of data obfuscation based on XOR-Masking depends on the specific program to obfuscate. XOR-Masking with constant mask bring negligible execution time and memory overhead. When an opaque mask is used the program slows down on average of 4–5 times and it requires 20% more memory. In case the opaque mask is cached, the slowdown is on +12% on average and the memory overhead is negligible. When the mask is not cached, the execution time overhead is correlated with the number of accesses to obfuscated fields, that dynamically occur in an execution scenario.

Table 4
Memory overhead (MEM) caused by XOR-Masking obfuscation.

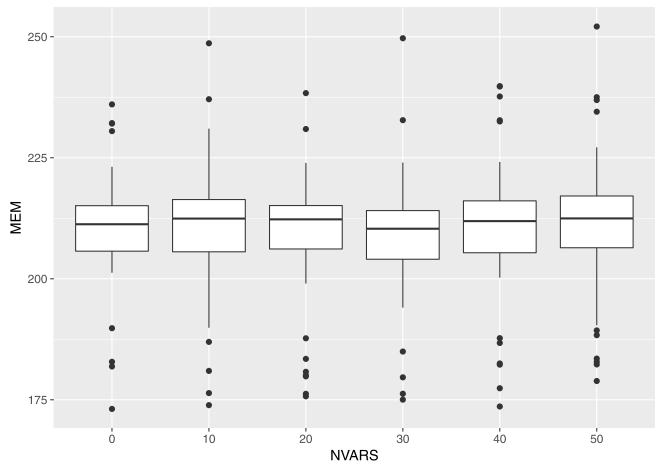
Application	Clear	Constant	Opaque	Opaque cached
Java2word	209.66	207.99 (+0%)	210.52 (+0%)	209.43 (+0%)
Jfreechart	264.96	264.41 (+0%)	295.45 (+11%)	259.45 (-2%)
Xml-Security	888.90	877.52 (+0%)	961.79 (+8%)	901.45 (1%)
Jimfs	345.45	345.49 (+0%)	339.7 (-1%)	342.66 (+0%)
Joda-Time	239.53	238.19 (+0%)	345.58 (+44%)	237.63 (-1%)
Stream-lib	163.68	163.34 (+0%)	164.19 (+1%)	165.81 (1%)
Gson	234.78	235.20 (+0%)	290.24 (+23%)	237.03 (+1%)
cglib	231.90	230.08 (+0%)	289.98 (+25%)	247.68 (+7%)
Quartz	956.35	95582(+0%)	1 360.14 (+42%)	885.55 (-7%)
LibRec	1 742.49	1 744.83 (+0%)	1 734.28 (0%)	1 745.98 (+0%)
jackson	841.07	837.36 (+0%)	1 615.42 (+92%)	682.11 (-18%)
GraphHopper	247.94	250.15 (+1%)	255.58 (+3%)	246.61 (+0%)

Table 5
Analysis of correlation between static/dynamic field access and time/memory overhead.

Obfuscation variant	Time overhead			Memory overhead		
	No. of fields	Static refs	Dynamic refs	No. of fields	Static refs	Dynamic refs
Constant	0.138	0.326	**0.048**	0.399	0.613	0.804
Opaque	0.417	0.804	**$<$ 0.001**	0.530	0.960	0.974
Opaque cached	0.545	0.584	0.826	0.457	0.887	0.885



(a) Time



(b) Memory

Fig. 8. Runtime overhead when increasing the complexity of the *k*-clique problem used for opaque constants.

5.6. RQ$_3$: Impact of the k-clique problem complexity

By increasing the complexity of the *k*-clique problem used in data obfuscation, we can turn static analysis arbitrarily hard, however at an increasing cost execution time overhead of the obfuscated code. In other words, the more obscurity corresponds to the larger obfuscation cost. RQ$_3$ is meant to quantify the practical impact of obfuscation on real world programs.

Practically, we pick one random case study application among those considered previously (i.e., Java2word) and we repeat the overhead measurement when obfuscation uses increasing values of *k*. Here we consider the obfuscation variant with the highest runtime cost, i.e. XOR-Masking with opaque mask.

We generate 5 distinct 3SAT propositional formula with increasing complexity, i.e. with increasing number of propositional variables: NVARS = {10, 20, 30, 40, 50} and $k = \lfloor 4.3 \cdot \mathrm{NVARS} \rfloor$. These propositional formulas are, then, reduced to distinct *k*-clique problems and, eventually, used to obfuscate the case study with 5 distinct obfuscation configurations.

All the test cases of this case study have been executed on the original clear code (without obfuscation) and on all the 5 obfuscated versions, collecting execution time and memory consumption. Measurement is repeated 100 times to control random measurement errors.

Experimental results are shown in Fig. 8. The boxplot on the left-hand side shows the execution time for clear code (left-most box) and for increasing complexity of obfuscation. While static analysis complexity grows exponentially with the size of the problem (*k*-clique is an NP-hard problem), we can observe a linear trend in the recorded execution time.

Average execution time values are also reported in Table 6. While the execution of clear code takes on average 0.31 seconds (first line), obfuscated code takes from 0.47 seconds (+52%) up to 0.91 seconds (+196%) depending on what obfuscation is used.

Table 6
Average execution time for clear and obfuscated program, with increasing obfuscation complexity.

NVARS	K	ETIME	Increase
-	-	0.31	-
10	43	0.47	+52%
20	86	0.58	+88%
30	129	0.69	+123%
40	172	0.80	+162%
50	215	0.91	+196%

Fig. 8, right-hand side shows the boxplot of memory used by the clear and obfuscated code with increasing NVAR (from left to right). We can notice that the memory required to run all the versions is approximately the same.

Considering these results, we can answer RQ$_3$ as follows:

While execution time overhead increases linearly with obfuscation complexity, no noticeable memory overhead is observed.

5.7. Possible extensions

As discussed in Section 2.1, our attack model focuses on static analysis. As such, we acknowledge that our approach is vulnerable to dynamic analysis. Potential extensions to overcome this limitation are represented by the following:

- *Integration with anti-debugging.* A debugger could be used to suspend the execution after the mask is computed and to read the mask value directly from the program memory. To overcome this limitation, our approach should be paired with an existing anti-debugging approach (Abrath et al., 2016), so that it would be much harder to attach a debugger and set breakpoints. An approach that integrates data obfuscation and anti-debugging might mitigate at the same time either malicious reverse engineering based on static analysis and attacks based on dynamic analysis;

- *Dynamic opaque predicates.* Once the mask leaks, it can be used to attack all the instances of an obfuscated program, because all of its executions share the same mask. Dynamic opaque predicates Palsberg et al. (2000); Xu et al. (2016) are predicates whose value changes at each execution. Control flow obfuscation has been built on top of dynamic opaque predicates such that, even if predicate values are different in each execution, the control flow and the semantic of the obfuscated code is preserved. We could extend our approach using a similar concept, i.e. data obfuscation based in dynamic opaque predicates. According to this intuition, the mask should be different for each execution, but the same clear value of sensitive variables should correctly decoded when needed by the program logics. In this way, even if the mask value would leak, it would be of limited benefit for mounting a reusable attack, because subsequent executions of the obfuscated program would be based on different values of the mask, that would be still unknown to the attacker.

6. Related work

The most related work by Moser et al. (2007) and our previous work (Tiella and Ceccato, 2017). Moser at al. presented an approach based on based on the 3SAT problem to craft opaque constants that are difficult to guess by static analysis. However, their work has the main limitation that obfuscation can not be too resilient, because the obfuscator has the solve the same 3SAT problem as the static analysis attacker. In our previous work, we overcome this limitation by proposing opaque constants based on the *k*-clique problem, that can be checked fast by the obfuscator but still require exponentially longer time by an attack based on static analysis.

Other relevant techniques to tackle the problem of generating opaque constants and opaque predicates were presented by Collberg et al. (1997, 1998); Collberg and Thomborson (2002) and by Wang et al. (2011).

The techniques proposed by Collberg et al. leverage the hardness (undecidability, in general Ramalingam, 1994) 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. (2011) 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 study runtime impact of our obfuscation scheme.

Work by cryptographers, such as Barak et al. (2012), 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 (Garg et al., 2013). 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. (2009) 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 and Thomborson (2002) as a quality related to how difficult is an obfuscated program to be automatically de-obfuscated. Karnick et al. (2006) measure resilience as the number of errors generated when decompiling the obfuscated code. Sutherland et al. (2006) relied on a program binary instrumentation tool to measure the fraction of the obfuscating transformations that attackers can undo automatically. Udupa et al. (2005) 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.

7. Conclusions and future work

Opaque constants are cornerstone features to extend and improve existing obfuscation transformations. We presented a novel extension of XOR-Masking data obfuscation, by using opaque constants to hide the mask. Our novel approach to data obfuscation shown to be sound and experienced performance overhead that can be controlled by trading off obscurity code performance.

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

Abrath, B., Coppens, B., Volckaert, S., Wijnant, J., De Sutter, B., 2016. Tightly-coupled self-debugging software protection. In: Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering. ACM, p. 7.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K., 2012. On the (im) possibility of obfuscating programs. J. ACM (JACM) 59 (2), 6.

Cannell, J., 2013. Obfuscation: Malwares best friend.

Ceccato, M., Tonella, P., Basile, C., Coppens, B., De Sutter, B., Falcarin, P., Torchiano, M., 2017. How professional hackers understand protected code while performing attack tasks. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). IEEE, pp. 154–164.

Ceccato, M., Tonella, P., Basile, C., Falcarin, P., Torchiano, M., Coppens, B., De Sutter, B., 2019. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. Emp. Softw. Eng. 24 (1), 240–286.

Collberg, C., Thomborson, C., Low, D., 1997. A Taxonomy of Obfuscating Transformations. Technical Report 148. Dept. of Computer Science, The Univ. of Auckland.

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

Collberg, C.S., Thomborson, C., 2002. Watermarking, tamper-proofing, and obfuscation: tools for software protection. IEEE Trans. Softw. Eng. 28, 735–746. doi:10.1109/TSE.2002.1027797.

Falcarin, P., Collberg, C., Atallah, M., Jakubowski, M., 2011. Guest editors' introduction: software protection. Software, IEEE 28 (2), 24–27.

Garey, M.R., Johnson, D.S., 2002. Computers and intractability, 29. wh freeman New York.

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

Jakubowski, M.H., Saw, C.W., Venkatesan, R., 2009. Iterated transformations and quantitative metrics for software protection.. In: SECRYPT, pp. 359–368.

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

Karp, R.M., 1972. 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. Springer US, Boston, MA, pp. 85–103. Reducibility among Combinatorial Problems

Moser, A., Kruegel, C., Kirda, E., 2007. Limits of static analysis for malware detection. In: Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual. IEEE, pp. 421–430.

Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q., Zhang, Y., 2000. Experience with software watermarking. In: Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00). IEEE, pp. 308–316.

Ramalingam, G., 1994. The undecidability of aliasing. ACM Trans. Program. Lang. Syst. (TOPLAS) 16 (5), 1467–1471.

Selman, B., Mitchell, D.G., Levesque, H.J., 1996. Generating hard satisfiability problems. Artif. Intell. 81 (1), 17–29.

Sutherland, I., Kalb, G.E., Blyth, A., Mulley, G., 2006. An empirical examination of the reverse engineering process for binary files. Computers & Security 25 (3), 221–228.
Tiella, R., Ceccato, M., 2017. Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 182–192.
Udupa, S.K., Debray, S.K., Madou, M., 2005. Deobfuscation: Reverse engineering obfuscated code. In: Proceedings of the 12th Working Conference on Reverse Engineering. IEEE Computer Society, Washington, DC, USA, pp. 45–54. doi:10.1109/WCRE.2005.13.
Wang, Z., Ming, J., Jia, C., Gao, D., 2011. Linear obfuscation to combat symbolic execution. In: Computer Security–ESORICS 2011. Springer, pp. 210–226.
Wroblewski, G., 2002. General Method of Program Code Obfuscation. Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002. Ph.D. thesis..
Xu, D., Ming, J., Wu, D., 2016. Generalized dynamic opaque predicates: A new control flow obfuscation method. In: International Conference on Information Security. Springer, pp. 323–342.
Zarate, C., Garfinkel, S.L., Heffernan, A., Gorak, K., Horras, S., 2014. A Survey of XOR as a Digital Obfuscation Technique in a Corpus of Real Data. Technical Report. DTIC Document.

Roberto Fellin obtained the Master degree in Computer Science from University of Trento in 2019 with a thesis on au- tomated data obfuscation. After graduation, Roberto started his professional career as software developer in a banking software company in Trento.

Mariano Ceccato is tenured researcher in FBK (Fondazione Bruno Kessler) in Trento, Italy. He received the PhD in Com- puter Science from the University of Trento in 2006 with the thesis ”Migrating Object Oriented code to Aspect Oriented Programming”. He is author or coauthor of more than 70 re- search papers published in international journals, conferences and workshops. He was recently visiting research scientist in the Software Verification and Validation Laboratory Centre for ICT Security, Reliability, and Trust (SnT), University of Luxembourg. His research interests include security testing, empirical software engineering, code hardening, reverse engineering and re-engineering. He was program co-chair of the 12th IEEE Working Conference of Source Code Analysis and Manipulation (SCAM 2012), held in Riva del Garda, Italy.