

Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis

Ramtine Tofighi-Shirazi, Irina Mariuca Asavoae, Philippe Elbaz-Vincent, Thanh-Ha Le

▶ **To cite this version:**

Ramtine Tofighi-Shirazi, Irina Mariuca Asavoae, Philippe Elbaz-Vincent, Thanh-Ha Le. Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis. 3rd International Workshop on Software PROtection, Nov 2019, London, United Kingdom. hal-02269192

HAL Id: hal-02269192
https://hal.science/hal-02269192

Submitted on 30 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis

Ramtine Tofighi-Shirazi
Univ. Grenoble Alpes, CNRS, Institut Fourier
Trusted Labs, Thales Group
Meudon, France
ramtine.tofighishirazi@thalesgroup.com

Irina Măriuca Asăvoae
Trusted Labs, Thales Group
Meudon, France
irina-mariuca.asavoae@thalesgroup.com

Philippe Elbaz-Vincent
Univ. Grenoble Alpes, CNRS, Institut Fourier
F-38000 Grenoble, France
philippe.elbaz-vincent@univ-grenoble-alpes.fr

Thanh-Ha Le
Work done when author was at Trusted Labs
Meudon, France
lethanhha.work@gmail.com

ABSTRACT

We present a new approach that bridges binary analysis techniques with machine learning classification for the purpose of providing a static and generic evaluation technique for opaque predicates, regardless of their constructions. We use this technique as a *static automated deobfuscation tool* to remove the opaque predicates introduced by obfuscation mechanisms. According to our experimental results, our models have up to 98% accuracy at detecting and deobfuscating state-of-the-art opaque predicates patterns. By contrast, the leading edge deobfuscation methods based on symbolic execution show less accuracy mostly due to the SMT solvers constraints and the lack of scalability of dynamic symbolic analyses. Our approach underlines the efficiency of hybrid symbolic analysis and machine learning techniques for a static and generic deobfuscation methodology.

KEYWORDS

Obfuscation, deobfuscation, machine learning, symbolic execution, opaque predicate

1 INTRODUCTION

Automatic program analysis is widely used in research and industries for various software evaluation purposes. In this context, software reverse engineering, which consists in the understanding of the internal behavior of a program, relies on various program analyses techniques such as static or dynamic symbolic execution. To prevent the application of software reverse engineering techniques, code obfuscation [11, 41] is a broadly employed protection methodology which transforms a program into an equivalent one that is more difficult to understand and analyze. Among these obfuscation mechanisms, opaque predicates represent one of the major and fundamental obfuscation transformations used by obfuscators to mitigate the risk of reverse engineering. Opaque predicates represent constant branching conditions that are obfuscated for the purpose of hiding the fact that they are constant. Thus, an opaque predicate value is known a-priori by the defender, but shall be hard to deduce for an attacker. We choose this obfuscation transformation for its variety of types and constructions and their common use as foundation for other obfuscation transformations as means of improving the transformations robustness and resiliency. Opaque predicates [12] are widely used as technique for various security applications, *e.g.* metamorphic malware mutation [9], Android applications [27] or white-box cryptographic implementations. As a consequence, several works focus on the deobfuscation of opaque predicates (*e.g.* [5, 7, 8, 16, 29, 32, 42]) in order to evaluate the quality of the obfuscated code rendered by this transformation. However, these techniques are often based on dynamic analysis and are therefore limited or not scalable.

Problem setting: Existing state-of-the-art opaque predicates deobfuscation techniques and tools suffer from the following limitations:

(1) *Specificity:* Techniques that evaluate opaque predicates are often focused on specific constructions, hence lacking of generality towards all existing patterns of such obfuscation transformation.
(2) *Code coverage:* Most recent deobfuscation techniques are based on dynamic symbolic execution which require the generation of instruction traces. The ability to cover all paths of the program is an issue that prevents, in some cases, the complete code deobfuscation.
(3) *Scalability:* Dynamic symbolic execution techniques may also lack of scalability on some types of code such as malwares that use specific triggers (*e.g.* an input from a Command and Control server) to execute the non-benign part of the code. Thus, dynamic analysis may not scale and cover the non-triggered part of the code.
(4) *Satisfiability modulo theories solvers:* SMT solvers used in path-reachability analyses suffer from several limitations depending on the constructions of the opaques predicates. Some constructions that are based on aliases or mixed boolean and arithmetic expressions usually cause SMT solvers to fail at predicting the feasibility of a path.

Our work has the goal to re-introduce static analysis for obfuscated software evaluation and deobfuscation. To this end, we propose a new approach that bridges static symbolic execution and machine learning models to provide a generic evaluation of opaque predicates.

We present several studies and experiments towards the construction of machine learning models that can either detect an opaque predicate or predict its invariant value without executing the code. We also extend our design to the deobfuscation of such obfuscation transforms, regardless of their constructions, by creating a static analysis plug-in within a widely used reverse engineering tool called IDA [19]. To further evaluate our methodology, we compare it against available static and dynamic symbolic-based tools for the deobfuscation of opaque predicates. We conduct further evaluations against obfuscators such as Tigress [10] and OLLVM [23].

The aftermath of our work shows that combining machine learning techniques with static symbolic analysis provides a generic, automatic, and accurate methodology towards the evaluation of opaque predicates. Our work shows that machine learning enables a better efficiency and genericity for this application, while allowing us to work without SMT solvers to predict reachable paths.

Contributions:

(1) We present our novel methodology that binds binary analysis and machine learning techniques to evaluate and deobfuscate opaque predicates statically. A presentation of several studies towards an efficient and accurate creation of machine learning models is also given.
(2) The evaluation of our methodology against state-of-the-art obfuscators such as Tigress and OLLVM, as well as novel opaque predicate constructions such as *bi-opaque* predicates.
(3) The illustration of the efficiency of our methodology, used as a static analysis deobfuscation tool, on several datasets by comparing it to existing state-of-the-art deobfuscation tools based on symbolic execution and SMT solvers.

Our paper is organized as follows: in Section 2 we recall background information on opaque predicates types, constructions, and deobfuscation methods. Then we introduce some notions of supervised machine learning. In Section 3, we present our methodology which combines binary analysis and machine learning to achieve an efficient evaluation and deobfuscation of opaque predicates. Section 4 presents our experiments towards an accurate model construction, whereas Section 5 illustrates our evaluations on state-of-the-art publicly available obfuscators. A comparison to existing symbolic-based deobfuscation techniques against our methodology is also provided in Section 5.3. We then describe our design limitations and perspectives in Section 6, along with related work in Section 7, before concluding in Section 8.

2 TECHNICAL BACKGROUND

In this section we briefly present opaque predicates, obfuscation and deobfuscation techniques, and introduce several notions related to supervised machine learning.

2.1 Code obfuscation

Collberg *et al.* [11] define code obfuscation as follows:

Let $P \xrightarrow{T} P'$ be a transformation T of a source program P into a target program P'. We call $P \xrightarrow{T} P'$ an *obfuscating transformation* if P and P' have the same observable behavior. Consequently, the following conditions must be fulfilled for an obfuscating transformation : if P fails to terminate, or terminates with an error condition, then P' may or may not terminate; otherwise, P' must terminate and produce the same output as P. Several obfuscation transformations exist, each of them having their own purpose: obfuscate the layout, the data, or the control-flow of a program. A classification of all these obfuscations, as well as known deobfuscation methods with their different purposes, has been provided by S. Schrittwieser *et al.* [39].

2.2 Opaque predicates

Often combined with bogus code, opaque predicates [12] aim at encumbering control-flow graphs with redundant infeasible paths. Compared to other control-flow obfuscation transformations such as control-flow flattening or call/stack tampering [28], opaque predicates are supposedly more stealthy (*i.e.* hard for an attacker to detect) because of the difficulty to differentiate an opaque predicate from original path conditions in a binary code. In the followings, we give an overview of some existing types and constructions of opaque predicates.

2.2.1 Opaque predicate types. We denote by ϕ a predicate, *i.e.* a conditional jump, within a binary code. Such predicate can be evaluated to both *true* or *false* (*i.e.* 0 or 1). We denote by O the obfuscation function that generates opaque predicates. O takes as input a predicate ϕ and produces the opaque predicate $O(\phi)$. For security purposes, $O(\phi)$ should be stealthy (indistinguishable from any other ϕ) and resilient (its value should not be easily known by an attacker). There are two types of opaque predicates, namely the *invariant* and the *two-ways*. C. Collberg *et al.* define these predicates by, respectively, P^T, P^F, and $P^?$. Our methodology aims at detecting and deobfuscating opaque predicates of types P^T and P^F. Next we explain the introduced notations P^T, P^F and $P^?$.

Invariant opaque predicates: Let $O(\phi) : X \rightarrow \{0, 1\}$ be an obfuscated predicate that evaluates to either 0 or 1 and O be the function that obfuscates the predicate. We denote by X the set of all possible inputs. If for all $x \in X$, $O(\phi)(x) = 1$ (resp. 0), then we say that the predicate is **always true** (resp. **always false**), denoted P^T (resp. P^F). These opaque predicates are said *invariant*, as they always evaluate to the same value for all possible inputs.

Two-ways opaque predicates: Another type of opaque predicates is referred to as *two-way*, which can evaluate to both true or false for all possible inputs. Such a construction requires both branches to be semantically equivalent in order to preserve the functionality of the code that will be executed. In other words we have, if for all $x \in X$, $Pr_{x \leftarrow X}[O(\phi)(x) = 1] = \frac{1}{n}$ with $n \in \mathbb{N}^+$, then the predicate is **either true or false**, regardless of the input x.

2.2.2 Opaque predicate constructions. Several proposals exist in the literature about how to construct a resilient and stealthy opaque predicate, *e.g.* [30, 44]. Each of these constructions has the purpose to thwart specific deobfuscation analyses and will be summarized in Section 2.3.

Arithmetic-based opaque predicates. Constructed using mathematical formulas which are hard to solve, they aim at encoding invariants into arithmetic properties on numbers.

Mixed-boolean and arithmetic based opaque predicates. Otherwise known as MBA [45], they are based on a data encoding technique using linear identities involving boolean and arithmetic operations. The resulting encoding is made dependent on external inputs such that it cannot be deobfuscated by compiler optimizations.

Alias-based opaque predicates. They are one of the first choices of Collberg *et al.* [12] for their construction. Aliasing is represented by a state of a program where certain memory location is referenced to by multiple pointers, and pointer alias analysis is undecidable.

Environment-based opaque predicates. These constructions use constant elements lifted from the system, or libraries.

Bi-opaque opaque predicates. Bi-opaque constructs aim at employing the weaknesses of symbolic execution, and are introduced in recent work [44]. Based on the observation that deobfuscation techniques using symbolic execution are prone to some challenges and limitations, bi-opaque predicates intend to either introduce false negatives or false positives results. Such construction has been shown effective against state-of-the-art deobfuscation tools based on dynamic symbolic execution, such as Triton [1] or Angr [40].

2.3 Deobfuscation

Due to their wide utilization and popularity, opaque predicates are target of several published attacks. Each of these deobfuscation methodologies has strengths and limitations as summarized in Table 1.

A first deobfuscation technique called *probabilistic check* consists in executing n times a program segment to see if a predicate is invariant. Such technique can be combined with fuzzing on the inputs. However, it is prone to high false positives and negatives results while depending on the possibility to execute n times the code.

Also, due to the overhead introduced by most complex opaque predicates constructs, it has been showed in the literature that there are surprisingly relatively few predicates that are used over and over again. This leads to a possible *pattern matching attack* (*i.e.* dictionary attack) [16], where one takes obfuscated predicates from a program being attacked and pattern-matches the source code against known examples. Nevertheless, it is possible to build variants of opaque predicates that cannot be matched using dictionary attacks, which implies a high false negative rate.

Another technique that uses *abstract interpretation* [32] provides correctness and efficiency in the deobfuscation of some specific constructions of opaques predicates. It consists in a static symbolic attack that can be only efficient against some classes of invariant arithmetic-based opaque predicates, but does not focus on other types and structures.

Another recently introduced technique [8] uses *program synthesis*. Originally designed for the deobfuscation of virtualized code, this approach has been successful for the simplification of MBA expressions.

Moreover, current state-of-the-art deobfuscation approaches use *automated proving* to compute if a predicate is opaque [5, 29].

Udupa *et al.* [43] use static path feasibility analysis based on SMT solvers to determine the reachability of execution paths. Their methodology inherit the limitations of static analysis, namely path explosion. This is the reason why recent automated proving techniques are based on dynamic symbolic execution (*i.e.* DSE) to check path feasibility or infeasibility [5] and remove opaque predicates. Yet, automated proving based analysis, either static or dynamic, suffers from the SMT solvers restrictions. It has been showed that SMT solvers fail against MBA opaque predicates, alias-based constructions, and can even be misguided by more recent constructions such as bi-opaque predicates.

Overall, DSE is currently considered the most effective methodology against opaque predicates, but the evaluation of such technique has been shown effective mainly against arithmetic or environment based opaque predicates. This demonstrates the importance of a generic and scalable methodology that can evaluate both stealth and resilience of opaque predicates for all existing constructions.

2.4 Supervised Machine Learning

The decision of labeling a predicate as opaque, and even more as invariant P^T or P^F opaque predicate, can be considered as classifications problems. Our target is to find algorithms that work from external supplied instances (*e.g.* binaries, instructions traces, etc.) in order to produce general hypotheses. From these hypotheses, we want to make predictions about future instances. *Supervised machine learning* provides a dedicated methodology that achieves this goal. The aim of a supervised machine learning is to build a *classification model* which will be used to assign *labels* to testing or unknown instances. In other words, let X be our inputs (*i.e.* instances) and Y the outputs (*i.e.* predicted labels). A supervised machine learning algorithm will be used to learn the mapping function f such that $Y = f(X)$. The goal is to approximate f such that for any new instance X we can predict its label Y. In our case the inputs are represented by n-dimensional vectors of numerical features that represent these features, *i.e. features vectors*, for which the extraction is described in the following paragraph.

2.4.1 Feature extraction. In the machine learning terminology, the inputs of a model are usually derived from what is called *raw data*, *i.e.* the data samples we want to classify or predict. These data samples cannot be directly given to a classification model and need to be processed beforehand. This processing step is called *feature extraction* and consists in combining the raw data variables into numerical features. It allows to effectively reduce the amount of data that must be processed, while accurately describing the original dataset of raw data. In our case, since raw data are text documents (*e.g.* disassembly code, symbolic execution state, etc.), one practical use of feature extraction consists in extracting the *words* (*i.e.* the features) from the raw data and classify them by frequency of use (*i.e.* weights). Different approaches exist for understanding what a word is and to compute its weight. In this paper we use the *bag of words* approach which identifies terms with words. As for the weights, we studied *term frequency* (*i.e.* how frequently a word occurs in a document) with and without *inverse document frequency* [22] used in Section 4 in order to select the best possible extraction technique.

Constructions	Probabilistic check	Pattern matching	Abstract interpretation	Automated proving	Program synthesis
Arithmetic-based	✓[43] High FN/FP	✗ High FN	✓[32]	✓[5, 29]	✗
MBA-based	✗ High FN/FP	✓[16]	✗	✗ (limitations of SMT solver)	✓[7, 8]
Alias-based	✗ High FN/FP	✗	✗	✗ (limitations of symbolic execution)	✗
Environment-based	✗ High FN/FP	✗	✗	✓[5, 29]	✗
Bi-opaque	✗ High FN/FP	✗	✗	✗ High FN/FP	✗

Table 1: Illustrations of opaque predicates deobfuscation strengths and targets.

2.4.2 Classification algorithms. The choice of which specific learning algorithm to use is a critical step. Many classification algorithms exist [21], each of them having different mapping functions. Classification is a common application of machine learning. As such, there are many metrics that can be used to measure and evaluate the accuracy as well as the efficiency of our models. In order to compute these metrics, *k-Fold Cross-Validation* is a frequent technique [25]. It consists in reserving a particular set of samples on which the model does not train. This limited set of samples allows to estimate how the model is expected to perform on data not used during the training phase. The parameter k refers to the number of groups that a given dataset of samples is split into, in order to calculate the mean of our models *accuracy* as well as the *F1 score* based on the value of k. While the accuracy of the model represents the ratio of correctly predicted labels to the the total of labels, F1 score takes both false positives and negatives into account. In our experimentations and evaluations, the accuracies and F1 scores are calculated using *k-fold cross-validation*, with $k = 20$ for a better generalization of our model to unknown instances.

3 OUR METHODOLOGY

Our methodology design is built in two parts. The first part consists in creating a machine learning model for the evaluation and deobfuscation of opaque predicates. The second part uses the validated model in order to remove such obfuscation transformation statically. Figure 1 illustrates our methodology. The first step consists in generating a set of obfuscated binaries. Our datasets of C code samples are presented in Section 4.1. In the second step, the binary is disassembled and we collect and labelize each predicate, *e.g.* defining if the predicate is opaque or normal, as described in Section 3.1. . The third step consists in a depth-first search algorithm to collect each path leading to a predicate. We use a thresholded static symbolic execution to collect our raw data for the machine learning model. These data are normalized, processed and used to train and validate our model in a fourth step, as presented in Section 3.2. Finally, the fifth and final step shows that our model can be used and integrated in a static deobfuscation tool to predict and remove opaque predicates as presented in Section 5.

3.1 Binary analysis

Our methodology relies on static symbolic execution to retrieve *the semantics of the predicate constructions* before the machine learning classification models evaluates them. Thus, a first step in our design is the generation of *raw data*. This refers to a representation of data samples that contain noisy features and need to be processed in order to extract informative characteristics from the data samples, before training a model. Since our goal is to evaluate the opaque predicates, we choose to generate our raw data from the disassembled binary code control-flow graph.

Moreover, in order to have a *scalable methodology*, we work statically in order to prevent the need of executing the code. This approach also permits a *better code coverage* compared to existing dynamic approaches. However, our approach can be extended to instruction traces in cases where the analyzed code is encrypted or packed. The raw data used contains the symbolic expressions $\mathcal{S}$ of collected predicates ϕ denoted by $\mathcal{S}_\phi$.

We studied different formats and contents of such raw data as well as their impact on the efficiency of the trained model (see Section 4). In the following sections we present the binary analysis part of our design, namely *thresholded static symbolic execution*, which we employ to generate the raw data from predicates.

3.1.1 Thresholded Static Symbolic Execution. Static symbolic execution is a binary analysis technique that captures the semantics (*i.e.* logic) of a program. An interpreter is used to trace the program, while assuming symbolic values for inputs rather than obtaining concrete values as a normal execution would. A symbolic state $\mathbb{S}$ is built and consists in a set of symbolic expressions S for each variables (*i.e.* registers, memory, flags, etc.). Several techniques exist for symbolic execution [3].

In our work we use disassembled functions to collect the symbolic expressions of a predicate $\mathcal{S}_\phi$. We start by generating all possible paths from a function entry point to a predicate ϕ using a depth-first search algorithm. The latter prevents us from using SMT solvers to generate all feasible paths since they are prone to limitations and errors depending on the protections applied. In order to avoid path explosion, we use a *thresholded* static symbolic execution technique that bounds the number of paths generated for one predicate and the amount of time the analysis has to iterate on a loop. Note that our methodology is intra-procedural

since publicly available obfuscators, *e.g.* Tigress, O-LLVM, generate intra-procedural opaque predicates.

Path generation: We denote by ϕ_n the n-th predicate within a disassembled function F in a binary B. When a predicate is identified, we generate all paths from F entry point to the collected ϕ_n using a depth-first search (*i.e.* DFS) algorithm. DFS expands a path as much as possible before backtracking to the deepest unexplored branch. This algorithm is often used when memory usage is at a premium, however it remains hampered by paths containing loops. Thus, we use two distinct thresholds, one for loop iterations denoted by α_{loop}, and one for the number of paths to be generated denoted α_{paths}.

Symbolic state generation: In order to have a symbolic state, we use all collected paths of a predicate. We denote by $\mathcal{P}$ the set of all collected paths σ of a predicate ϕ. Let S be the symbolic execution interpreter function such that $S(\sigma_i) = \mathbb{S}_{\phi^{\sigma_i}}$. In other words, the symbolic execution interpreter S returns a symbolic state $\mathbb{S}_{\phi^{\sigma_i}}$ for a path σ_i, $i \in [O, |\mathcal{P}|]$ of a predicate ϕ. The generated symbolic states for all predicates will be used as raw data and then be processed for the classification models.

3.2 Machine learning

We experiment different instances for our classification models to study the impact on their accuracy. Since symbolic execution is often based on an intermediate representation that captures all the semantics as well as side effects of the assembly instructions, several intermediate representations exist and are widely used, *e.g.* LLVM-IR or Miasm-IR [14]. We implemented our methodology using Miasm2 reverse engineering framework, which integrates translators from Miasm-IR to other languages (*e.g.* SMT-LIBv2 [6], Python [37], or C [24]). This gives us the ability to study the impact of the language used to express the symbolic expressions, within our raw data, on our classification models.

3.2.1 Raw data. Intermediate representations use concrete values within their generated expressions. This causes raw data to depend on addresses that are specific to some binaries and prevents our models to scale on unknown data. Listing 1 illustrates this issue with one predicate symbolic expression in the Miasm-IR language. Moreover, some intermediate representations, *e.g.* Miasm-IR, use identifiers in order to express modified registers name or memory locations. This may further affect the scalability of our trained models.

For the purpose of having a model that can scale to unknown data, we use a normalization phase that replaces identifiers and concrete values by symbols, and non-alphanumerical characters by alphanumerical words. This is a necessary step for a complete features extraction phase that sometimes excludes non-alphanumerical characters when working on text-based raw data. In Listing 1 lines 10 and 13 we provide examples of the normalization step.

```
1 # Miasm-IR predicate expression before normalization
2 # Miasm-IR predicate expression of an P^T opaque predicate
3 ExprId('IRDst', size=64) = ExprInt(0x402b36, 64)
4
5 # Miasm-IR predicate expression of an P^F opaque predicate
6 ExprId('IRDst', size=64) = ExprInt(0x402209, 64)
7
8 # Miasm-IR predicate expression after normalization
9 # Normalized Miasm-IR predicate expression of an P^T opaque
      predicate
10 ExprId(id1, size=64) = ExprInt(v1, 64)
11
12 # Normalized Miasm-IR predicate expression of an P^F opaque
      predicate
13 ExprId(id1, size=64) = ExprInt(v1, 64)
```

Listing 1: Miasm-IR predicates expressions before and after normalization

Since our methodology computes a full symbolic state from any function entry-point to a targeted predicate, there is a need to know if all information within the collected symbolic state is relevant for our models. The goal is to have many features for an accurate classification without adding too much noise.

Another issue to be avoided is having raw data samples that do not contain enough information to distinguish between samples that have different labels, as illustrated also in Listing 1. In other words, we may have two expressions that are identical but have different labels, *e.g.* the first being the expression of a P^T and the second an expression of a P^F. To avoid this matter we use the thresholded symbolic execution, which generates expressions for each path leading to a predicate. Listing 2 illustrates the predicates expressions from Listing 1 along with others memory and registers expressions from their symbolic state. Namely, line 10 in Listing 1 corresponds to lines 2-5 in Listing 2, while line 13 in Listing 1 corresponds to lines 8-11 in Listing 2. We can see that now we have more informations that allows us to distinguish between both predicates.

```
1 # Normalized Miasm-IR predicate expression of an P^T opaque
      predicate
2 ExprId(id1, size=64) = ExprInt(v1, 64)
3 ExprId(id2, size=1) = ExprInt(v2, 1)
4 ...
5 ExprId(id9, size=1) = ExprInt(v5, 1)
6
7 # Normalized Miasm-IR predicate expression of an P^F opaque
      predicate
8 ExprId(id1, size=64) = ExprInt(v1, 64)
9 ExprId(id2, size=1) = ExprCond(ExprMem(ExprId(v3, size=64), size
      =8), ExprInt(v4, 1), ExprInt(v3, 1))
10 ...
11 ExprId(id9, size=1) = ExprInt(v3, 1)
```

Listing 2: Miasm-IR predicate expressions after our normalization phase

We study the use of several expressions in our raw data to distinguish between sample that have different labels. To this end, we divide our instances into three sets:

- **Set 1**: with samples containing only the expression of the predicate in a static single assignment form (*i.e.* SSA) as illustrated in Listing 1.
- **Set 2**: with samples containing only the expressions of the predicate and its corresponding flags in a SSA form.
- **Set 3**: with samples containing the full symbolic state of a path, from an entry-point to a targeted predicate, *i.e.* all memory, flags, and registers modified in a SSA form as illustrated in Listing 2.

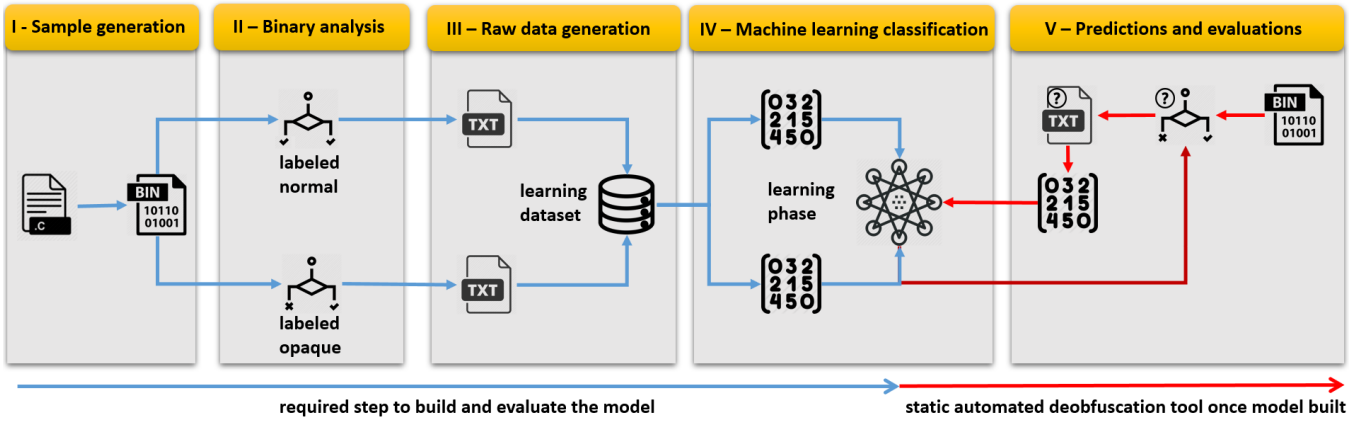


Figure 1: Evaluation steps of opaque predicates.

In Section 4.2, each set is studied in order to find the best possible raw data content. We start by calculating for each set the similarity percentages based on 5000 samples of predicates, either *normal* or *opaque* predicates generated by the Tigress obfuscator on a dataset of C code samples (see Section 4.1). In other words, we search for raw data with different labels (*e.g.* P^F and P^T) but with the same content. As we can see in Table 2, only the Set 3 has a low rate of similarities between opaque or legit raw data content (3.5%) and between P^T and P^F raw data (6%). This indicates that Set 3 is more

Raw data	Detection similarities	Deobfuscation similarities
Set 1	24.94%	31.92%
Set 2	17.38%	26.62%
Set 3	**3.5%**	**6%**

Table 2: Percentage of our raw data content similarities for each sets.

suited for our raw data representation.

3.2.2 Decision tree based models. Decision trees [36] predict the output by learning simple decision rules deduced from the training dataset. The internal nodes of a decision tree contain binary conditions based on input features vectors, whereas the leaves are associated with the class labels we want to predict. Decision trees are built recursively. The root node contains all the training instances and each internal node splits its training instances into two subsets according to a condition based on the input. Leaf nodes however represent a classification or decision on these training instances. Different approaches exist for the splitting conditions of internal nodes [18]. However, one downside of decision tree models is *over-fitting* [15] which may cause the creation of over-complex trees that do not generalize the data well. In our case, the decision tree model is capable of identifying and deobfuscating an opaque predicate $O(\phi)$. We choose to create two distinct models: a first one that evaluates the stealth of an opaque predicate and a second one to evaluate its resiliency, as presented in the following paragraphs.

Model for stealth (detection). The construction of a classifier consists in the definition of a mapping function $C_f : \mathcal{D} \rightarrow [0, 1]$ that, given a document d (*i.e.* an input), returns a *class label*, which is represented by a number (here 0 or 1) that defines the category of d. Applied to the evaluation of opaque predicates stealthiness, the function can be seen as $C_f : \mathcal{D} \rightarrow$ [NORMAL, OPAQUE]. In other words, given the term-frequency vector of a symbolic execution state D, from a function entry point to a predicate, our model mapping function C_f will return two values: NORMAL or OPAQUE. If a model is capable of detecting a predicate as opaque, we can assume that the transformation is not stealthy.

Model for resiliency (deobfuscation). In order to evaluate the resiliency of an opaque predicate, we construct a model with a different function as presented for the evaluation of stealthiness. Indeed, our goal is to predict if an opaque predicate is of type P^T or P^F, thus, the function $C_f : \mathcal{D} \rightarrow [0, 1]$ in that context can be expressed as $C_f : \mathcal{D} \rightarrow$ [TRUE, FALSE].

The choice of the best suited classification algorithm is often made on accuracy but in our work we choose our model based on its *transparency* to easily interpret our results. Since many learning algorithms exist, the next section will present our experiments to select the best classification model for both detection and deobfuscation of opaque predicates.

4 EXPERIMENTS

In this section we present our study of efficient and accurate creation of classification models. We start by introducing the datasets variety used in our work.

4.1 Datasets

Our experiments are made on several C code samples. We use the scikit-learn API [31] for the implementation of the models. The datasets contain various types of code, each of them having different functionalities in order to have a model that does not fit to a specific type of program, as listed below:

- GNU core utilities (*i.e.* core-utils) binaries [33] for normal predicate samples;
- Cryptographic binaries for obfuscated and non-obfuscated predicates [13];
- Samples from [4] containing basic algorithms (*e.g.* factorial, sorting, etc.), non-cryptographic hash functions, small programs generated by Tigress;
- Samples involving the uses of structures and aliases [2, 20].

Our choice is motivated by their low ratio of dependencies and their straightforward compilation. This makes their obfuscation possible using tools such as Tigress and OLLVM without errors during compilation. A list of all different combinations of obfuscation transformations and options related to Tigress is given in Appendix A and Listing 18.

Dataset size determination: One important point is to determine the amount of samples required since this can significantly impact the cost of our studies and evaluations, as well as the reliability of our results. If too much samples are collected, we face a longer evaluation time but if there are not enough samples is our dataset, our results may be irrelevant. Several propositions based on statistical tests allow to determine the size of our datasets depending on the area of research [17]. Based on these works, we create our datasets with between 5000 and 15.000 samples in order to have a high probability of detection and of confidence level. Each of our datasets are *balanced*, *i.e.* with an equal number of samples of each classes. Next, we present our studies using these datasets.

4.2 Preliminary studies

The goal of our experiments is to investigate and answer the following questions:

- **Study 1:** Which raw data language is the more efficient (in terms of time and space) and also the more accurate?
- **Study 2:** Which raw data content best expresses the normal and opaque predicates?
- **Study 3 and 4:** Which classification model is more accurate and which feature extraction algorithm is best suited?

The following paragraphs present our experiments for each question. For this section and for our evaluations (see Section 5) we used a laptop running Windows 7 with 16 GB of RAM and a Intel Core i7-6820HQ vPro processor.

Study 1: Raw data language selection. Our goal is to select the most appropriate language for the symbolic execution engine. We use Miasm-IR, which we compare with the translators it implements in SMT-LIBv2 language, C, and Python. After normalizing these languages, as presented in Section 3.2.1, we use our dataset of normal predicates from core-utils binaries along with structured-based opaque predicates from Tigress to study several points:

(1) Which set of samples is more efficient in terms of disk space?
(2) Which set of samples is more efficient in terms of computation time?
(3) Which language is more accurate for our models when representing our raw data?

Table 3 illustrates our experiments using 20-fold cross-validation on decision-tree based models. For each language, we used a dataset of 10000 balanced samples.

Raw data language	Miasm2	SMT-LIBv2	C	Python
Detection accuracy (%)	94%	90%	87%	87%
Deobfuscation accuracy (%)	88%	80%	78%	78%
Execution time for detection (s)	15s	114s	21s	20s
Execution time for deobfuscation (s)	12s	50s	15s	13s
Size of dataset (GB)	1.91GB	37.4GB	2.11GB	1.98GB

Table 3: Study of the raw data language accuracy and efficiency

We observe that Miasm2 intermediate representation gives higher accuracy rates for both the detection and deobfuscation model. Moreover, it is more efficient in terms of disk space used (as opposed to the SMT-LIBv2 dataset), which leads to a faster time of execution. This is mainly due to the fact that Miasm2 intermediate language has a small set of terms expressing the semantics of the code as compared to other languages in our study. According to these results, we choose Miasm2 for all of our raw data samples for the remaining of the paper.

Study 2: Raw data content selection. It remains to single out the most suitable content that will express the construction of normal and invariant opaque predicates. Table 2 in Section 3.2.1 shows that the use of full symbolic state representation prevents having similarities between samples of different classes (*i.e.* labels). Thus, based on the same dataset of core-utils and structured-based

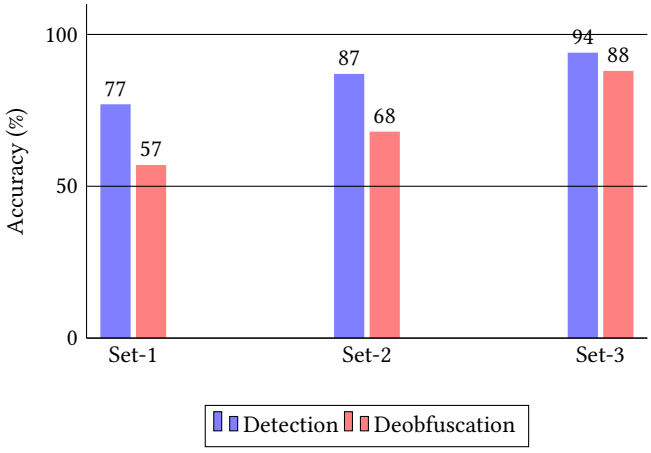


Figure 2: Predictions accuracy on the different raw data sets

opaque predicates generated with Tigress, we measure the average of our models accuracies for both detection and deobfuscation, evaluated with a 20-fold cross-validation. Figure 2 confirms that the Set 3, *i.e.* the full symbolic state, presents a better accuracy for both detection (at 94%) and deobfuscation (at 88%) when using the decision tree algorithm on balanced datasets of 10000 samples.

Study 3 and 4: Classification algorithm and feature extraction selection. In order to properly evaluate our methodology, we need to select the appropriate features extraction techniques combined with an accurate classification algorithm.

We have done experiments with the most common classifications models [26], namely decision trees, k-nearest neighbors, support vector machines, neural network, naive Bayes, and random forest. The use-case of our experiments is to evaluate the stealth of structured-based opaque predicates generated with Tigress on our datasets. The features are expressed using *term-frequency* (*i.e.* tf) vectors as well as *td-idf* vectors in order to compare both extraction techniques. Default parameters are applied for each classification algorithms used in our study.

Classification algorithm	Term-frequency vectors	TD-IDF vectors
Decision-tree	**94%**	**93%**
k-Nearest Neighbors	91%	92%
Support Vector Machine	87%	71%
Linear Support Vector Machine	77%	83%
Multi-layer Perceptron	84%	92%
Multinomial Naive-Bayes	58%	75%

Table 4: Accuracies of different classification models using tf and td-idf vectors.

Table 4 illustrate our results. We can observe that the decision tree model stands out from others when term-frequency vectors are used. It averages 94% of detection accuracy whereas *k*-Nearest Neighbors averages 91%. As for the use of *td-idf* vectors, the decision tree model has a better accuracy at 93%. *According to this experiment, we choose the* ***Decision-tree*** *classification algorithm with* ***term-frequency*** *as features extraction technique in our methodology.*

5 EVALUATIONS

Our goal in this section is to evaluate opaque predicates stealth and resiliency using a model based on decision trees. We divide our evaluation into two parts:

(1) *Stealth:* can the model differentiate an opaque predicate from a normal predicate, ***i.e.* is the opaque predicate stealthy?**
(2) *Resilience:* can the model differentiate a P^T opaque predicate from a P^F opaque predicate, ***i.e.* is the opaque predicate resilient?**

5.1 Measuring stealth

In this section we focus on the evaluation of stealthiness of opaque predicates. Namely, we want to see if our model is able to distinguish opaque predicates from normal predicates. Based on our datasets, our goal is to measure the efficiency of our model for the detection of opaque predicates based on different constructions. *Note that each datasets is balanced and contains 10000 samples.*

Tigress: The Tigress obfuscator can generate a variety of complex obfuscation transformations, *e.g.* MBA-based, structured-based or environment-based. To this end, we use several datasets of different opaque predicates constructions, balanced with normal predicates, to evaluate our model for detection. *Dataset 1* contains arithmetic, MBA and environment-based opaque predicates whereas *Dataset 2* contains structured-based (*i.e.* alias-based) opaque predicates. Moreover, we used a third dataset (*Dataset 3*) that combines these opaque predicates with other obfuscation transformations such as arithmetic, literal, and data encodings (*i.e. EncA*, *EncL*, and *EncD*, respectively) joined with control-flow flattening (*Flat*) and code virtualization (*Virt*). Our results are illustrated in Table 5.

Datasets	Types of OP	Other transforms	Analysis time	Accuracy(%)	F1 Score(%)
Dataset 1	Arithmetic, Environment-based	None	1.13 s	93 %	93 %
Dataset 2	Arithmetic, Structure-based	None	2.14 s	95 %	95 %
Dataset 3	Arithmetic, MBA, Structure-based	EncA, EncL, EncD, Flat, Virt	1 s	99 %	98 %

Table 5: Evaluations of stealth (detection) using Tigress

Regardless of their types and of the implication of other obfuscation transformations, our detection model is able to efficiently predict if a predicate is *opaque* or *normal*. Indeed, the detection of arithmetic and environment-based opaque predicates scores an accuracy and F1 score of 93%, whereas arithmetic and structured-based opaque predicates are less stealthy for our model with scores up to 95%. However, as more obfuscation techniques are combined with opaque predicates, our predictions accuracy and F1 score rises to respectively 99% and 98%. This is due to the fact that opaque predicates patterns, once combined to other combination of transforms, become more specific thus lower their stealthiness. In our case however, code virtualization (*i.e.* Virt) is applied before opaque predicates, as illustrated in Appendix A. The opposite, namely applying code virtualization after other transformations, is a limitation to our methodology since the generated opaque predicates will be virtualized, thus transformed into byte-code.

OLLVM:. In order to evaluate our model against opaque predicates generated by OLLVM, we split our evaluations in two sets. The first set uses samples obfuscated only with opaque predicates (*i.e.* the bogus control-flow transformations *bcf*). The second set uses samples obfuscated with opaque predicates combined with control-flow flattening and instructions substitutions (*i.e. fla* and *sub*, respectively) to see if we can evaluate opaque predicates stealthiness when they are combined with others transformations. Table 6

Datasets	Types of OP	Other transforms	Analysis time	Accuracy(%)	F1 Score(%)
Dataset 1	Arithmetic-based	None	2 s	89 %	89 %
Dataset 2	Arithmetic-based	fla, sub	1 s	95 %	94 %

Table 6: Evaluations of stealth using OLLVM

illustrates our results. In the second dataset, our model is able to efficiently detect the labels of most predicates. However, when opaque predicates are not combined with other obfuscation transformation, we observe a loss of efficiency, from 95% to 89% accuracy. This indicates that OLLVM opaque constructions are stealthier than other constructs, thus our model cannot properly distinguish them from normal predicates. At best, it will requires more training samples for our model in order to have a better accuracy. One reason for their stealthiness in regard to our model is the fact that OLLVM arithmetic opaque predicates are bloc-centric, with basic encodings, which may have similar patterns to normal predicates from

hash functions or cryptographic codes in our datasets. However, when they are combined to the other transformations, their patterns become more specific and our model has better prediction results.

Bi-opaque: Several constructions exist for bi-opaque predicates, among which float-based (*i.e.* using floating instructions) or symbolic-memory based. We use their obfuscator based on the OLLVM framework to evaluate our detection model. As we can see in Table 7,

Datasets	Types of OP	Other transforms	Analysis time	Accuracy(%)	F1 Score(%)
Dataset 1	Floats	None	0.6 s	93 %	93 %
Dataset 2	Symbolic-memory	fla, sub	0.9 s	98 %	98 %

Table 7: Evaluations of stealth using Bi-opaque predicates from [44]

our model is efficient at detecting bi-opaque predicates with 93% accuracy for float-based constructs. Bi-opaque predicates are constructed based on the same patterns as OLLVM opaque predicates but using floating-point instructions and registers instead. However, symbolic-memory based constructs rely on more specific patterns, thus allowing a better detection rate at 98% accuracy and F1 score.

5.2 Measuring resiliency

Once a predicate is detected as being *opaque*, our goal is to measure its resiliency. In other words, we want to know if our model is able to deobfuscate, *i.e.* predict the output of the opaque predicate. Our evaluations are based on invariant opaque predicates, P^T and P^F, generated using different constructions.

Tigress: The patterns between P^T and P^F are more difficult to predict since both predicates are opaque and generated using the same construction. However, the underlying invariant properties render our models efficient towards their deobfuscation. Table 8

Datasets	Types of OP	Other transforms	Analysis time	Accuracy(%)	F1 Score(%)
Dataset 1	Arithmetic, Environment	None	0.3 s	90 %	91 %
Dataset 2	Arithmetic, Structure	None	1 s	88 %	87 %
Dataset 3	Arithmetic, MBA, Structure	EncA, EncL, EncD, Flat, Virt	3 s	92 %	92 %

Table 8: Evaluations of resiliency (deobfuscation) using Tigress

shows our results. We can observe that our model is able to detect environment-based invariants with scores of 90% accuracy and 91% of F1 score on balanced datasets of 5000 samples. For structure-based invariants, we get slightly lower results, with 88% and 87% of accuracy and F1 score. This is due to the fact that structured-based invariants use aliasing, producing patterns which are less dissimilar than for environment-based opaque predicates. However, our model has a better accuracy and F1 score (92% for both) when other transformations are used. Thus, we are able to efficiently and accurately predict the invariant value of opaque predicates generated with Tigress, regardless of their constructions, and of the combination of obfuscation transformations used.

Bi-opaque, OLLVM, and Tigress: Since OLLVM only produces P^T opaque predicates, we choose to combine all available samples generated from our three evaluated obfuscators. A first dataset is used to evaluate our deobfuscation models against normal predicates and opaque predicates generated without any other transformations. A second dataset is used to combined opaque predicates with others existing transformations from these obfuscators. *Note*

Datasets	Types of OP	Other transforms	Analysis time	Accuracy(%)	F1 Score(%)
Dataset 1	Arithmetic, MBA, Environment, Structure, Symbolic-memory, Floats	None	1 s	92 %	91 %
Dataset 2	Arithmetic, MBA, Environment, Structure, Symbolic-memory, Floats	fla, bcf, EncA, EncL, EncD, Flat, Virt	0.5 s	95 %	95 %

Table 9: Evaluations of resiliency using Bi-opaque, OLLVM, and Tigress

that all datasets are balanced and contain 15000 samples. Our results in Table 9 show that our methodology is efficient against all patterns of opaque predicates from available obfuscators. Our model is able to detect the invariant patterns of all the opaque predicate constructs with 92% accuracy and 91% F1 score. Moreover, when these opaque predicates are combined with other obfuscation transformations, the scores rise up to 95%.

5.3 Deobfuscation methodology

Our methodology can be used as an efficient deobfuscation technique, if it is based on an adequate dataset of training samples. We developed our methodology as an experimental IDA [19] plug-in that detects directly on the disassembled binary any opaque predicates and deobfuscates them, if needed. We will compare our results with existing opaque predicates deobfuscation tools based on SMT solvers and symbolic execution, such as DROP [35]. The latter is an IDA Pro plug-in based on Angr, which uses static symbolic execution for the removal of invariant and contextual opaque predicates. Meanwhile, for the dynamic symbolic execution, we use Miasm2 dynamic symbolic execution engine. We employ several datasets of opaque predicates obfuscated with various constructions and transformations. Moreover, we remove all samples used in our evaluations datasets from our learning samples used to built our model.

Our invariant opaque predicates are generated mainly from [4] and Table 10 shows the results. For each deobfuscation tool we use several samples obfuscated by different obfuscators (*c.f.* column Obfuscator) and obfuscation transformations (*c.f.* Obfuscation). Column "OP detection rate" indicates the percentage of removed opaque predicates, whereas column "#FP, #FN" shows the number of false positive and false negative results respectively. Finally column "Errors" indicates if an error occurred during the analysis, *e.g.* lack of memory or a timeout.

We observe that, for a static analysis, our experimental plug-in performs better at removing opaque predicates with complex constructs such as the one generated by Tigress, or the bi-opaque constructs. We obtain better results than the experimental plug-in DROP, as well as a better rate than DSE-based techniques for most constructions of opaque predicates.

Tool	Obfuscator	Obfuscation	OP detection rate %	#FP, #FN	Errors
DROP	OLLVM	bcf	100%	1,0	0
	OLLVM	bcf, sub	100%	0,0	1
	Bi-opaque	float	100%	4,0	0
	Bi-opaque	symbolic-memory	75%	1,5	2
	Tigress	Environment-based	60%	1,8	0
	Tigress	Structure-based	25%	2,12	1
	Tigress	MBA, struct	10%	0,10	8
Our methodology	OLLVM	bcf	100%	0,0	0
	OLLVM	bcf, sub	100%	0,0	0
	Bi-opaque	float	92%	0,0	0
	Bi-opaque	symbolic-memory	100%	1,0	0
	Tigress	Environment-based	88%	2,3	0
	Tigress	Structure-based	82%	1,4	0
	Tigress	MBA, struct	85%	2,2	0
Miasm DSE	OLLVM	bcf	100%	0,0	0
	OLLVM	bcf, sub	100%	0,0	0
	Bi-opaque	float	100%	0,0	0
	Bi-opaque	symbolic-memory	85%	0,3	0
	Tigress	Environment-based	88%	1,2	0
	Tigress	Structure-based	65%	1,7	0
	Tigress	MBA, struct	52%	2,10	6

Table 10: Comparisons of opaque predicates deobfuscation using machine learning vs. SMT-solver based analyses.

6 LIMITATIONS AND PERSPECTIVES

Our experiments and evaluations underline the efficiency of decision tree models to detect and deobfuscate opaque predicates. The most important achievement of our technique is that it allows a generalization to most invariant opaque predicates constructions. Next we enumerate the limitations of our method.

The first limitation is due to decision tree models and the switch between obfuscators. Namely, we can observe that a model that learns from samples generated using one obfuscator, cannot efficiently fit to transformations of another obfuscator if they use different kinds of constructions. This also hinders our ability to detect new constructions of opaque predicates.

A second limitation comes from the use of static symbolic execution to generate the symbolic state as a raw data. Such process is part of the deobfuscation application of our methodology, and, as any static analysis, may be time consuming. This explains the use of our thresholded static symbolic execution in order to prevent as much as possible issues such as path explosion.

Our work proposes a new application of machine learning techniques for the purpose of evaluating obfuscation transformations, and also for removing them in a static automated manner. Our experimentations and evaluations, indicate that our design can be extended to other complex constructions of opaque predicates such as thread-based and hash-based constructs. Future work include also a more in-depth study of obfuscation transforms combinations and options, as well as the generation of deobfuscated program to report any good or bad behaviors (*e.g.* crashes).

7 RELATED WORK

Many binary analysis techniques are often based on pattern matching for either detecting plagiarism, or malicious behaviors. Recent studies show the efficiency of machine learning and deep learning techniques for the detection and classification of malwares, *e.g.* [34], which also implicates the detection of similar codes within the malwares samples. More closely related to the obfuscation area, the work in [38] aims at recovering meta-data information using machine learning techniques. Their goal is to detect the obfuscation transformation used in several protected binaries generated by Tigress. Their evaluations show that naive Bayes and decision tree models can be efficient at detecting obfuscation transformations using filtered instruction traces. However, their work focuses on the recovery of informations about the obfuscation techniques used, but it does not aim at deobfuscating.

Another work, [4], aims at predicting the resiliency of obfuscated code against symbolic execution attacks. They use machine learning to measure the ability of several different symbolic execution engines to run against various layers and combinations of obfuscation techniques. Nevertheless, machine learning is not primarily used to remove any obfuscation transforms.

To summarize, existing work shows that machine learning techniques are pertinent *w.r.t.* of the classification or the detection of features within binary samples. However, to the best of our knowledge, no deobfuscation study and methodology exists regarding these techniques. For this reason, in this paper, we proposed an efficient way to evaluate both the stealth and the resilience of opaque predicates through several studies and experiments combining binary analysis technique and machine learning.

8 CONCLUSION

In this paper we applied machine learning techniques to the evaluation of opaque predicates. By introducing the different constructions of opaque predicates and the limitations from dynamic symbolic execution techniques and SMT solvers, we underlined the importance of studying other alternatives for generic evaluations of these transformations.

We proposed a new approach that bridges a thresholded static symbolic execution with machine learning classification to evaluate both the stealth and resilience of invariant opaque predicates constructions. The use of static symbolic execution allows us to have a better code coverage and scalability, which combined with a machine learning model, permits a generic approach by discarding the use of SMT solvers. Our studies illustrate that our choices conduct towards the implementation of an efficient and accurate evaluation framework against state of the art obfuscators. We created two models for the evaluation of stealth and resiliency of state-of-the-art opaque predicates constructions, with results up to 99% for detection and 95% for deobfuscation. Moreover, we extended our work to a deobfuscation plug-in and compared our results to other tools, showing the efficiency of machine learning for the deobfuscation of most invariant opaque predicates constructions. As future work, we propose to extend machine learning techniques to the evaluation of other obfuscation transformations as well as a more in-depth study of deep learning techniques, which we envision to render promising results.

We believe that our work provides a new framework to evaluate opaque predicates transformations, as well as a new alternative towards their static and automated deobfuscation.

ACKNOWLEDGMENTS

This work is supported by the French National Research Agency in the framework of the Investissements d'Avenir program (ANR-15-IDEX-02).

A TIGRESS COMMANDS

We used the following combinations of obfuscation transformations for our datasets. Note that the combinations listed in *italic* are considered as clean samples since they do not generate opaque predicates.

- AddOpaque (16 or 32 times)
- AddOpaque, EncodeLiterals
- *EncodeLiterals*
- AddOpaque, EncodeArithmetics
- EncodeArithmetics, AddOpaque
- *EncodeArithmetics*
- AddOpaque, EncodeData
- EncodeData, AddOpaque
- *EncodeData*
- AddOpaque, EncodeArithmetics, EncodeLiterals, EncodeData
- EncodeData, EncodeArithemtics, EncodeLiterals, AddOpaque
- AddOpaque, Flatten
- Flatten, AddOpaque
- *Flatten*
- *Flatten, EncodeData, EncodeArithemtics, EncodeLiterals*
- Virtualize, AddOpaque
- *Virtualize*
- *Virtualize, EncodeData, EncodeArithemtics, EncodeLiterals*
- *Virtualize, Flatten*
- Flatten, AddOpaque, EncodeData, EncodeArithemtics, EncodeLiterals
- Virtualize, AddOpaque, EncodeData, EncodeArithemtics, EncodeLiterals
- Virtualize, Flatten, AddOpaque, EncodeData, EncodeArithemtics, EncodeLiterals

A.1 Commands options

```
# AddOpaque options
tigress --Transform=InitEntropy --Transform=InitOpaque --
      InitOpaqueStructs=list,array,env --Functions=main --
      Transform=AddOpaque --Functions=${3} --AddOpaqueCount=${
      NUM} --AddOpaqueKinds=call,fake,true

# Flatten
tigress --Transform=Flatten --FlattenDispatch=switch,goto --
      Functions=${3}

# Virtualize
tigress --Transform=Virtualize --VirtualizeDispatch=switch,
      direct,ifnest,linear --Functions=${3}

# EncodeLiterals
tigress --Transform=EncodeLiterals --Functions=${3} --
      EncodeLiteralsKinds=integer

# EncodeArithmetics
tigress --Transform=EncodeArithmetic --Functions=${3} --
      EncodeLiteralsKinds=integer

# EncodeData
tigress --Transform=EncodeData --LocalVariables=${4} --
      EncodeDataCodecs=poly,xor,add --Functions=${3}
```

Listing 3: Tigress commands for sample generation

REFERENCES

[1] 2015. *Triton: A Dynamic Symbolic Execution Framework*. SSTIC.
[2] The Algorithms. [n. d.]. C. https://github.com/TheAlgorithms/C/. [Online; accessed 30-01-2019].
[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657
[4] Sebastian Banescu, Christian S. Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, USA*. 189–200. http://dl.acm.org/citation.cfm?id=2991114
[5] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *2017 IEEE Symposium on Security and Privacy, SP 2017, USA*. 633–651. https://doi.org/10.1109/SP.2017.36
[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.
[7] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. 2017. Effectiveness of synthesis in concolic deobfuscation. *Computers & Security* 70 (2017), 500–515. https://doi.org/10.1016/j.cose.2017.07.006
[8] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium, USENIX Security 2017, Canada*. 643–659. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko
[9] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2006 Proceedings, Germany*. 129–143. https://doi.org/10.1007/11790754$_8$

[10] Christian Collberg, Sam Martin, Jonathan Myers, Bill Zimmerman, Petr Krajca, Gabriel Kerneis, Saumya Debray, and Babak Yadegari. [n. d.]. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/index.html. [Online; accessed 30-01-2019].

[11] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. A Taxonomy of Obfuscating Transformations.

[12] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. 1998. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *POPL '98, USA*. 184–196. https://doi.org/10.1145/268946.268962

[13] Brad Conte. [n. d.]. crypto-algorithms. https://github.com/B-Con/crypto-algorithms. [Online; accessed 30-01-2019].

[14] Fabrice Desclaux. 2012. Miasm : Framework de reverse engineering. https://github.com/cea-sec/miasm. [Online; accessed 30-01-2019].

[15] Thomas G. Dietterich. 1995. Overfitting and Undercomputing in Machine Learning. *ACM Comput. Surv.* 27, 3 (1995), 326–327. https://doi.org/10.1145/212094.212114

[16] Ninon Eyrolles, Louis Goubin, and Marion Videau. 2016. Defeating MBA-based Obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection, SPRO@CCS 2016, Austria*. 27–38. https://doi.org/10.1145/2995306.2995308

[17] Rosa L. Figueroa, Qing Zeng-Treitler, Sasikiran Kandula, and Long H. Ngo. 2012. Predicting sample size required for classification performance. *BMC Med. Inf. & Decision Making* 12 (2012), 8.

[18] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. 2009. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*. Springer. http://www.worldcat.org/oclc/300478243

[19] Hex-Rays. [n. d.]. IDA Pro : Interactive DisAssembler. https://www.hex-rays.com/products/ida/index.shtml. [Online; accessed 30-01-2019].

[20] Simon Howard. [n. d.]. c-algorithms. https://github.com/fragglet/c-algorithms. [Online; accessed 30-01-2019].

[21] Mike James. 1985. *Classification Algorithms*. Wiley-Interscience, USA.

[22] Karen Spärck Jones. 2004. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation* 60, 5 (2004), 493–502. https://doi.org/10.1108/00220410410560573

[23] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, Brecht Wyseur (Ed.). IEEE, 3–9. https://doi.org/10.1109/SPRO.2015.10

[24] Brian W. Kernighan. 1988. *The C Programming Language* (2nd ed.). Prentice Hall Professional Technical Reference.

[25] Ron Kohavi. 1995. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Canada*. 1137–1145. http://ijcai.org/Proceedings/95-2/Papers/016.pdf

[26] Sotiris B. Kotsiantis. 2007. Supervised Machine Learning: A Review of Classification Techniques. *Informatica (Slovenia)* 31, 3 (2007), 249–268. http://www.informatica.si/index.php/informatica/article/view/148

[27] Aleksandrina Kovacheva. 2013. Efficient Code Obfuscation for Android. In *Advances in Information Technology - 6th International Conference, IAIT 2013, Thailand*. 104–119. https://doi.org/10.1007/978-3-319-03783-7$_1$0

[28] Arun Lakhotia, Eric Uday Kumar, and Michael Venable. 2005. A Method for Detecting Obfuscated Calls in Malicious Binaries. *IEEE Trans. Software Eng.* 31, 11 (2005), 955–968. https://doi.org/10.1109/TSE.2005.120

[29] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, USA, October 12-6, 2015*. 757–768. https://doi.org/10.1145/2810103.2813617

[30] Ginger Myles and Christian S. Collberg. 2006. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research* 6, 2 (2006), 155–171. https://doi.org/10.1007/s10660-006-6955-z

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[32] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. 2006. Opaque Predicates Detection by Abstract Interpretation. In *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Estonia*. 81–95. https://doi.org/10.1007/11784180$_9$

[33] GNU Project. 2002. GNU Core Utilities. https://www.gnu.org/software/coreutils/. [Online; accessed 30-01-2019].

[34] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. 2011. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security* 19, 4 (2011), 639–668.

[35] Thomas Rinsma. 2017. Seeing through obfuscation: interactive detection and removal of opaque predicates. https://github.com/Riscure/DROP-IDA-plugin. [Online; accessed 30-01-2019].

[36] Lior Rokach and Oded Maimon. 2014. *Data Mining With Decision Trees: Theory and Applications* (2nd ed.). World Scientific Publishing Co., Inc., USA.

[37] Guido Rossum. 1995. *Python Reference Manual*. Technical Report. Amsterdam, The Netherlands, The Netherlands.

[38] Aleieldin Salem and Sebastian Banescu. 2016. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, SSPREW 2016, USA, 2016*. 1:1–1:11. https://doi.org/10.1145/3015135.3015136

[39] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar R. Weippl. 2016. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1 (2016), 4:1–4:37. https://doi.org/10.1145/2886012

[40] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[41] Bjorn De Sutter, Cataldo Basile, Mariano Ceccato, Paolo Falcarin, Michael Zunke, Brecht Wyseur, and Jérôme d'Annoville. 2016. The ASPIRE Framework for Software Protection. In *Proceedings of the 2016 ACM Workshop on Software PROtection, SPRO@CCS 2016, Vienna, Austria, October 24-28, 2016*, Brecht Wyseur and Bjorn De Sutter (Eds.). ACM, 91–92. https://doi.org/10.1145/2995306.2995316

[42] Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, and Thanh Ha Le. 2018. DoSE: Deobfuscation based on Semantic Equivalence. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop, USA*. 1:1–1:12. https://doi.org/10.1145/3289239.3289243

[43] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. 2005. Deobfuscation: Reverse Engineering Obfuscated Code. In *12th Working Conference on Reverse Engineering, WCRE 2005, USA*. 45–54. https://doi.org/10.1109/WCRE.2005.13

[44] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael R. Lyu. 2018. Manufacturing Resilient Bi-Opaque Predicates Against Symbolic Execution. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg*. 666–677. https://doi.org/10.1109/DSN.2018.00073

[45] Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Information Security Applications, 8th International Workshop, WISA 2007, Korea*. 61–75. https://doi.org/10.1007/978-3-540-77535-5$_5$