Modeling Obfuscation Stealth through Code Complexity

Sebastian Schrittwieser[1], Elisabeth Wimmer[2], Kevin Mallinger[1], Patrick Kochberger[1,3], Caroline Lawitschka[1], Sebastian Raubitzek[2], and Edgar R. Weippl[1]

[1] University of Vienna, Austria, {firstname.lastname}@univie.ac.at
[2] SBA Research, Vienna, Austria {ewimmer,sraubitzek2}@sba-research.org
[3] St. Pölten University of Applied Sciences patrick.kochberger@fhstp.ac.at

Abstract. Code obfuscation is often utilized by authors of malware to protect it from detection or to hide its maliciousness from code analysis. Obfuscation stealth describes how difficult it is to determine which protection technique has been applied to a program and which parts of the code have been protected. In previous literature, most of the presented obfuscation identification methods analyze the program code itself (for example, the frequency and distribution of opcodes). However, simple countermeasures such as instruction substitution can have a negative impact on the identification rate. In this paper, we present a novel approach for an accurate obfuscation identification model based on a combination of multiple code complexity metrics. An evaluation with 4124 samples protected with 11 different obfuscations, combinations of obfuscations, and various compiler configurations demonstrates an overall classification accuracy of 86.5%.

Keywords: software obfuscation · stealth · code complexity

1 Introduction

Malware authors use various types of code obfuscations to make their code more difficult to detect and analyze. A Black Hat survey by Brosch et al. [4] from 2006 suggested that more than 90% of all malware samples identified in the wild use packing obfuscation to protect themselves from detection. In a more recent study from 2017, Rahbarinia et al. [33] found that 58% of all malware samples are protected with off-the-shelf packers (not taking into account custom packers which are used by about 35% of packed malware [27]). But also other obfuscation techniques, such as virtualization, data encoding/encryption, and library hiding, are frequently used to hide malicious code.

On the other hand, malware analysts and researchers aim to analyze unknown malware samples efficiently and understand how they work. An essential step in the analysis of obfuscated code is the identification of the obfuscation method. It significantly simplifies the analysis as tailored de-obfuscation methods often already exist. Obfuscating transformations complicate code merely in

its semantic representation and thus can often be undone if the exact obfuscation method (e.g., the obfuscator tool and its configuration) is known.

The stealth of an obfuscation indicates how opaque the application of an obfuscation is, both in terms of the type of obfuscation and the exact location in the code to which the obfuscation is applied. In previous literature, obfuscation stealth was mainly described in terms of code structures, such as for example, the frequency of opcodes. In contrast, this paper presents a novel approach that can detect obfuscation with high accuracy using a combination of different code complexity metrics. For our approach it is not required to know the relative changes in code complexity after applying an obfuscation. Instead, it is based purely on absolute values and the insight that the different obfuscations generate a characteristic pattern of increases, but also reductions of the various observed complexity metrics.

In particular, our main contribution in this paper is the introduction of a novel model for obfuscation stealth based on two obfuscators, 11 obfuscation combinations, and 4124 obfuscated programs.

The remainder of this paper is structured as follows. Section 2 discusses related work, while in Section 3 the fundamentals of code obfuscation, obfuscation stealth, and code complexity are presented. Section 4 introduces our novel approach to obfuscation identification. In Section 5, we show the results which we then discuss in Section 6. Section 7 concludes the paper.

2 Related work

In previous literature, a protection technique was described as stealthy if the resulting code resembles the original code as much as possible [8]. One major problem with quantifying stealthiness is that it highly depends on the original program whether or not a technique can be applied in a stealthy way. Sometimes a specific technique might produce code that fits perfectly into the original code. Other times however, the protection might generate code sections that clearly differ from the rest of the code, for example, in terms of its structure. Collberg et al. [29] described two types of obfuscation stealth. Local stealth measures the difficulty of identifying the exact location of an obfuscation applied to code, whereas steganographic stealth describes the difficulty of detecting that a specific obfuscation was applied at all.

Previous approaches to obfuscation detection are mainly based on code structures such as opcode frequencies. Kanzaki et al. [18] proposed an artificiality metric that measures the degree to which protected code can be distinguished from unprotected code. Their results showed that while some types of obfuscations strongly impact code artificiality, such as code encryption, others, for example control-flow modifying obfuscations such as CFG flattening, have a minimal effect. In 2017 Wang et al. [39] proposed a method for identifying the obfuscation tool, the applied obfuscation, and its configuration for protected Android applications. The method is based on machine learning using a feature vector from the Dalvik bytecode of the app. A related methodology was presented by Bacci

et al. [2] in 2018. It utilizes features extracted from the Smali representation of the application's bytecode.

LOM by Kim et al. [20] uses a neural network-based classifier on the opcode distribution of binary code for obfuscation identification.

3 Preliminaries

3.1 Code obfuscation

Obfuscating transformations convert code – either in the form of source, byte, or executable code – into code that is unintelligible in some form, either by a human code analyst or by an automated code analysis tool. The development of new code obfuscation techniques is mainly driven by the desire to hide the specific implementation of a program. This includes malware authors, who aim at hiding the malicious purposes of their code. Thus, breaking obfuscations is a fundamental prerequisite of malware detection. Code obfuscations can be categorized into various classes, such as layout transformations (which modify the superficial structure of the code) or control flow transformations (which alter the control flow path of a program while retaining its semantics [35]).

Obfuscations are usually applied to code through automatic tools. Some commercial source code protection solutions are on the market (e.g., Cloakware by Irdeto[4]), but also many freely available tools and online services. In the scientific community, the Tigress obfuscator [5] by Collberg et al. is widely used. Tigress was developed based on CIL [30] and MyJit[6] and is able to protect C source code with a variety of obfuscation methods. However, considerable uncertainty exists as to whether – and, if so, to what extent – the software protection is transferred to the binary program during compilation. Already in 2006, Madou et al. [23] were able to demonstrate empirically that not all types of protection survive the compilation process. To some extent, this undesired effect results from the fact that software protections intentionally make code more complicated, but a compiler attempts to generate efficient binary code through various optimization strategies. Thus, it often removes the protections or at least significantly reduces their strength.

In recent years, it was demonstrated that code obfuscation can also be applied to intermediate code representations during the compilation process after the optimizations have been conducted. With the Obfuscator LLVM (OLLVM) framework [17] it was prototypically demonstrated that compile-time protection of code is feasible.

In this work, we use obfuscations from both Tigress and Obfuscator LLVM to train our identification model. 6 different types of obfuscations and 4 combinations of obfuscation were used in our work. As one obfuscation is available in both obfuscators, a total of 11 obfuscation classes were defined. Table 1 describes the applied obfuscations.

[4] https://irdeto.com
[5] https://tigress.wtf
[6] https://myjit.sourceforge.net

Table 1. Applied obfuscations.

Technique	Obfuscator	Description
Opaque predicates	Tigress	Adds difficult to evaluate expressions to conditional jumps
Self-modifying code	Tigress	Adds code that modifies itself at runtime
Virtualization	Tigress	Transforms binary code to byte code of a custom virtual machine
Bogus control flow	OLLVM	Opaque predicates to make control flow less obvious; similar to Tigress opaque predicates
CFG flattening	both	Redirects all control-flow transfers to a central dispatcher
Instruction substitution	OLLVM	Replaces instructions with semantically equal ones

3.2 Obfuscation stealth

Obfuscation stealth describes how well-obfuscated code can be distinguished from untransformed code. Collberg et al. [7] have defined two different types of stealth. With local stealth, an attacker cannot determine a particular instruction as being affected by an obfuscating transformation. In contrast, with steganographic stealth, an attacker cannot determine whether a program has been transformed at all with a particular transformation or not.

At first glance, the coverage, i.e., how much code is actually modified, seems particularly relevant for the stealthiness of an obfuscation. In instruction substitution, for example, occurrences of certain instructions are replaced by semantically equivalent instructions or sequences of instructions. It is possible to specify how many of the occurrences are replaced. Coverage correlates with the number of code modifications. The smaller the coverage of an obfuscation, the smaller the modifications to the code.

However, the number of code modifications does not indicate how easily it can be distinguished from untransformed code. For example, a packer modifies the complete binary by encoding or encrypting the program's entire code as data. This fundamental structural modification of the binary seems more difficult to hide than protections with lower coverage. However, current literature proposes approaches such as using Huffman encodings [41] to make the packed code look structurally like actual binary code or shell code that looks like English prose [24]. While English prose can clearly be distinguished from actual shell code, the context where shell code is utilized makes it a perfect camouflage. Thus, coverage alone is not a good indicator of the stealthiness of an obfuscation.

3.3 Code complexity

As Collberg et al. [7] mentioned, code complexity metrics were initially created to help build reliable, readable, and maintainable software constructs. Generally, most of them can be summarized as describing a respective aspect of software,

typically textual, from which a complexity measure is derived. If this measure increases within a program, this program is then described as being more complex in relation to the measured properties. A property often utilized for this is obfuscation potency, i.e., how well humans are able to comprehend the code [9]. In contrast, obfuscation stealth has not been measured yet with code complexity metrics.

Over the last decades, various methodologies for code complexity measurements have emerged. For source code, the three most influential and widely used are the Halstead Complexity Metrics, Lines of Code, and Cyclomatic Complexity Metric [42, 19, 15]. These are also called *classic methods* as they were invented in the 1960s and 1970s; they are still highly relevant today, albeit primarily used in modified form and/or in tandem.

Halstead Complexity Metrics As an early pioneer of software science, Maurice Halstead was one of the first to quantitatively analyze software. His work resulted in the formalization of the *Halstead complexity metrics* [13], which consist of several sub-metrics.

- The *Halstead difficulty* measures how difficult it is to write or understand the code of a program. It is defined as $D = \frac{n_1}{2} \cdot \frac{N_2}{n_1}$, where n_1 is the number of distinct operators, n_2 is the number of distinct operands and N_2 is the total number of operands.
- *Halstead volume* estimates the required space for storing the program and is defined as $V = N \cdot \log_2 n$.
- *Halstead level* defines the implementation level $L = \frac{V_p}{V}$ where V_p is the potential or minimal volume $V_p = (2 + n_2) \cdot \log_2(2 + n_2)$.
- *Halstead effort* estimates the effort required for writing or understanding the program. It is defined as $E = D \cdot V$.
- *Halstead time* estimates the time required for writing the program and defined as $T = \frac{E}{18}$.

Lines of code The *Lines of Code* metric is a measure for the length of a program, whereby only executable lines are factored in [42, 37, 40, 14, 3]. Although being a basic metric, this brings several advantages, like being language-independent, fast to compute, and easy to comprehend [42]. The simplicity of the metric comes with several problems, though. For example, the content of the lines is completely disregarded so that a very simple and a highly complex line count as equal for the calculation. Furthermore, the program's structure is neglected with respect to jumping and branching. Finally, although size itself is not an immediate indicator of complexity within a software project, we can infer that larger programs contain more constructs and control structures and, therefore, more paths through the code. Thus, for the purpose of this work, the *Lines of Code* metric is considered a complexity metric.

Cyclomatic Complexity and Myer's interval *Cyclomatic Complexity*, the third of the classic metrics, describes the structure of software through the number of possible independent paths in its control flow graph [25, 16, 36, 10, 1, 34, 22]. This also functions as a measure of the nesting level and can be computed with binary code [5]. To calculate it, a flow graph G is created, and its cyclomatic value v is generated by $v(G) = e - n + 2p$. Here, e denotes the number of edges, n the number of nodes, and p the number of connected entities in G. This measure can be helpful in providing a good abstraction of the module's structure and works well in predicting future bugs, but it is completely blind to the length of a tested module. Because of extreme cases, like a single line of code potentially being able to have the same cyclomatic value as parts with hundreds of lines, Cyclomatic Complexity is often combined with other length-sensitive measures like Halstead metrics or Lines of Code. Similarly, *McCabe's cyclomatic complexity* [26] measures the complexity of a program by analyzing its control flow. The complexity $v(G) = E - N + 2p$ where $v(G)$ is the cyclomatic number of a graph G, E is the number of edges, N the number of nodes and p the number of connected components. *Myer's interval* [28] is an extension of McCabe's cyclomatic complexity. The Myer's interval $v(G) : v(G) + L$ adds the number of logical operators L to McCabe's cyclomatic complexity.

ABC metric Despite its traditional categorization as a size metric, the *ABC metric* [11] lends itself to the assessment of code complexity, given the quantitative focus on the evaluation of software components. Furthermore, the three components utilized within the *ABC metric* are fundamental constructs for any programming language, making them relevant in understanding the overall complexity of a software project. The three components, number of assignments (A), branches (B), and conditions (C), as a triplet, build the first representation (vector) of the ABC metric. The other possible representation is a number (Euclidean norm, L2 norm) calculated by the square root of the sum of the squared individual numbers: $|ABC| = \sqrt{A^2 + B^2 + C^2}$. Assuming there is at least one assignment, branch or condition, the ABC metric consequently is always a positive number $|ABC| > 0$.

Maintainability Index As the name already suggests, the *maintainability index* [31, 32, 6] was originally designed to measure how maintainable code is. It is based on the Halstead difficulty metric and is defined as $MIwoc = 125 - 10 \cdot \log(HE)$, where HE is the Halstead effort. A more complex variant is the *maintainability index without comments*, which combines the Halstead volume, McCabe's cyclomatic complexity, and LoC. It is defined as $MIwoc = 171 - 5.2 \cdot \ln(HV) - 0.23 \cdot CC - 16.2 \cdot \ln(LOC)$.

4 Approach

Depending on the obfuscation type, code complexity is affected to varying degrees. Furthermore, an obfuscation type does not contribute to all code com-

plexity metrics equally. For example, cyclomatic complexity is only increased by obfuscations that make structural changes at the control flow or call graph level. We also observed that obfuscations often significantly reduce individual code complexity metrics, thus the opposite of what would intuitively be expected. Our approach uses these characteristic patterns of increases and decreases in code complexity measurements to identify the implemented obfuscation technique. We discovered that absolute measurements are sufficient for identifying obfuscation techniques with ample accuracy and that relative changes from the original code are not required. To build our model, we assembled a set of 179 programs, which we then compiled in 52 different build and obfuscation configurations each. As the Tigress obfuscator can only handle single-file C source code, the set consists mainly of programs implementing one particular algorithm (e.g., hash function, sorting algorithm, units converter, etc.). Not all build and obfuscation configurations resulted in valid binary programs, as the obfuscators sometimes fail on specific program structures. The final set of training data contained 4124 binaries. We calculated the 14 code complexity metrics introduced in Section 3.3 for each. We combined eight from the initial 52 classes into a class *no-obfuscation* since all of them correspond to the different compilation variants of clang and gcc-musl without employing any obfuscation.

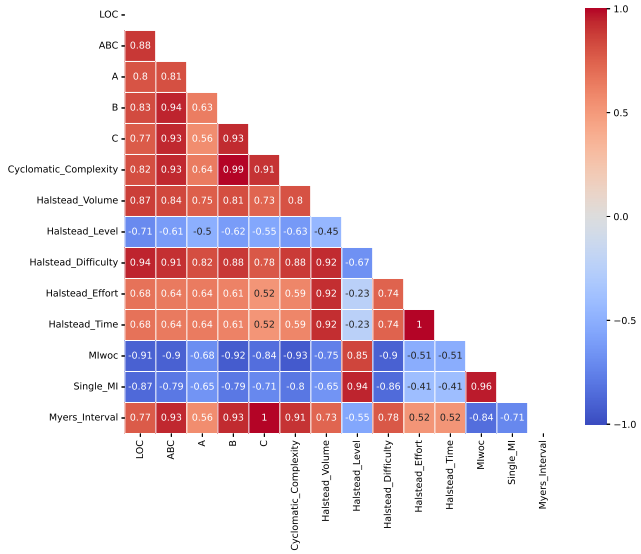

Fig. 1. Correlation coefficients for all complexity metrics.

We then pooled the classes corresponding to the same obfuscator and technique, i.e., combining the various optimization levels per obfuscation class. Thus, we ultimately ended up with 12 classes, where each obfuscation method has 300 to 336 observations, whereas we obtained 680 samples associated with no-obfuscation. We noticed that some metrics are highly correlated by analyzing the feature set. However, this is not surprising since several metrics are combinations or extensions of one another as described in Section 3.3. For example, the ABC metric is composed of A, B, and C, which are also part of the original feature set. We visualized the exact coefficients in the correlation matrix depicted in Figure 1.

We decided to fit models using both the whole feature set and a smaller set of values from five complexity metrics with a Pearson correlation coefficient $|\rho| \leq 0.9$. The features included in the smaller feature set were the following: Lines of Code, ABC, number of assignments (A), Halstead Volume, and Halstead Level. From here on, we refer to the respective feature sets as *All features* and *5/14 features*.

The following steps in preprocessing included shuffling the data and performing a stratified split into train and test data using the class proportions present. Here, we chose a split such that we ended up with 80% training data and 20% test data, with a corresponding `random_state = 42`. Finally, we standardized all features using the `StandardScaler`-function from `sklearn`, i.e., shifting the data to a mean of zero and a standard deviation of 1. As part of the hyperparameter search, we used 5-fold cross-validation to show the reliability of each model. We evaluated the models with respect to classification accuracy, F1 score, precision, and recall in order to get a holistic view of the model performances.

$$\text{Accuracy} = \frac{\text{Correct Classifications}}{\text{All Classifications}} \tag{1}$$

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2}$$

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \tag{3}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Negative} + \text{False Positive}} \tag{4}$$

After finding well-performing parameter sets for the machine learning models with each feature set respectively, we ran all models 100 times, using a new seed in each iteration. In addition to the best value of each performance metric, we also calculated the mean values over these runs to show the statistical validity of the results.

5 Evaluation

5.1 Model Selection

For the initial model selection, we employed the Lazy Learner framework `lazypredict` to get an overview of how different models perform on the data set under study.

Lazy Learner provides a fast way to test many different algorithms without further time-consuming parameterization. `ExtraTreesClassifier`, `LGBMClassifier`, and `RandomForestClassifier` performed best on our initial test with respect to the models' accuracy. Based on this result, we decided to investigate these three models further. To explore the potential of different algorithms in the context of obfuscation stealth, and because neural networks are known to perform well on classification problems [21, 12], we also included an `MLPClassifier` (Multi-Layer Perceptron). All applied models are part of `scikit-learn`, except for the `LGBMClassifier`, which is part of the `lightgbm` package.

5.2 Hyperparameter Tuning

For all models, we performed the hyperparameter tuning with Bayesian optimization using the `BayesSearchCV` function from `Scikit-optimize`. The sequential model-based optimization algorithm utilizes all previous loss observations in order to arrive at a well-performing set of parameters for the respective model. The incorporation of prior knowledge makes Bayesian optimization much more efficient than a grid search or random search [38]. We explored 100 parameter settings for each model, which we evaluated using a 5-fold cross-validation.

5.3 Model Results

As shown in Table 2, the prediction results on the test set were best for the `MLPClassifier`, achieving a maximum accuracy of 86.5% and an average accuracy of 83.9% over 100 runs respectively, using all features. The neural network outperforms the other classifiers, showing better results across all performance metrics. However, the `ExtraTreesClassifier`, the `LGBMClassifier`, and the `RandomForestClassifier` accomplish only slightly poorer results. In general, we observed that including all features in the model led to considerably better average performance, as seen from the accuracy values in Table 2 and Table 3. The severe difference in the accuracy scores implies that the nine complexity metrics we removed from the smaller feature set due to high correlation with other features carry information significant for the models.

Table 2. Maximum and average performance using all features for 100 iterations.

	Accuracy	**F1 Score**	**Precision**	**Recall**
Extra Trees	0.833 (⌀ 0.813)	0.833 (⌀ 0.813)	0.836 (⌀ 0.815)	0.833 (⌀ 0.813)
LGBM	0.834 (⌀ 0.806)	0.834 (⌀ 0.807)	0.837 (⌀ 0.810)	0.834 (⌀ 0.806)
Random Forest	0.827 (⌀ 0.800)	0.826 (⌀ 0.799)	0.828 (⌀ 0.801)	0.827 (⌀ 0.800)
MLP	**0.865** (⌀ 0.839)	**0.865** (⌀ 0.839)	**0.868** (⌀ 0.842)	**0.865** (⌀ 0.839)

The model specifications of the respective best models per classifier and feature set based on accuracy can be found in Table 4 in the Appendix A. The

Table 3. Maximum and average performance using 5/14 features for 100 iterations.

	Accuracy	**F1 Score**	**Precision**	**Recall**
Extra Trees	**0.770** (∅ 0.743)	**0.771** (∅ 0.742)	**0.773** (∅ 0.744)	**0.770** (∅ 0.743)
LGBM	0.752 (∅ 0.716)	0.752 (∅ 0.717)	0.755 (∅ 0.721)	0.752 (∅ 0.716)
Random Forest	**0.772** (∅ 0.729)	**0.772** (∅ 0.728)	**0.772** (∅ 0.730)	**0.772** (∅ 0.729)
MLP	0.764 (∅ 0.728)	0.764 (∅ 0.726)	0.766 (∅ 0.727)	0.764 (∅ 0.728)

boxplots in Figure 2 depict the accuracy ranges over 100 unique runs for all classifiers fitted on the data using all features. While Table 2 shows the superiority of the `MLPClassifier` concerning maximum accuracy, Figure 2 demonstrates where most predictions are located precisely. More than 50% of the predictions made by the `MLPClassifier` are better than all of the predictions made by the other three classifiers. Due to its good maximum accuracy and consistency over all runs, we conclude that using all features, the `MLPClassifier` is the best model choice when using all features.

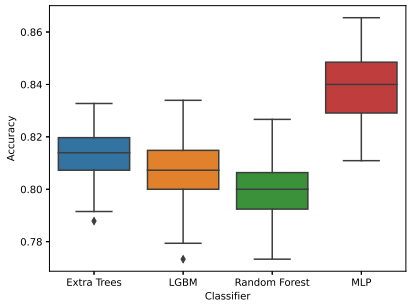

Fig. 2. Accuracy results over 100 runs for classifiers using all features.

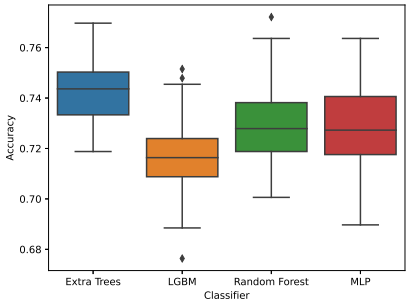

Fig. 3. Accuracy results over 100 runs for classifiers using 5/14 features.

Figure 3 shows that when using the smaller feature set consisting of only 5 out of the 14 complexity metrics, the performance of the classifiers decreases. The accuracy score corresponding to the multilayer perceptron classifier particularly suffers from excluding the strongly correlated variables. The best results were achieved by the `RandomForestClassifier` and the `ExtraTreesClassifier`, where the latter also performs best on average and therefore represents the most suitable model for our approach. The comparatively bad results achieved by the neural network might be due to MLPs relying on complex relationships and high-dimensional interactions between features. When removing features, the MLP model may lose more information as compared to the other models.

To further assess the performance of the selected best models, we constructed the normalized confusion matrices, which are depicted in Figure 4 and Figure 5

below. They visualize which classes are detected well and which techniques are often mistaken for one another.

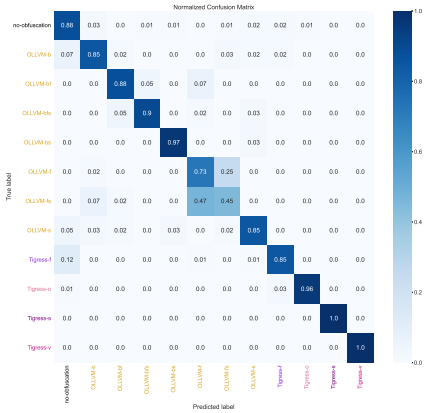

Fig. 4. Confusion matrix for results of the best model using all features.

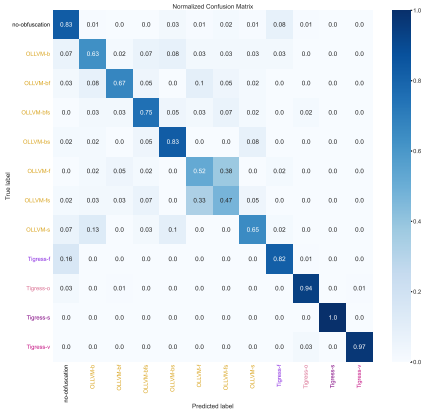

Fig. 5. Confusion matrix for results of the best model using 5/14 features.

On closer examination of the confusion matrix shown in Figure 4, we found that concerning Obfuscator-LLVM samples, the `MLPClassifier` has difficulties distinguishing between techniques that are combinations of one another. Most misclassifications occur between OLLVM-f (flattening) and OLLVM-fs, combining flattening and instructions substitution. There is a higher percentage of correct predictions for the classes corresponding to Tigress. In particular, the model can detect virtualization and self-modifying code exceedingly well. While some wrong classifications exist across compiler classes, most correspond to the same or related obfuscation methods. The model can generally differentiate between the two obfuscators, Tigress and Obfuscator-LLVM, and the no-obfuscation class quite well, showing only a few misclassifications. Indeed, most of those mistakes stem from the model wrongly classifying 12% of Tigress-f as no-obfuscation.

We also analyzed the normalized confusion matrix obtained when fitting the model using only 5/14 features. Looking at the results shown in Figure 5, the difficulties regarding the distinction of certain classes become more pronounced. In particular, differentiating between different techniques of the LLVM obfuscator seems to be harder for the model using only 5/14 features. Overall, the percentage of correct predictions decreases significantly. The drop in accuracy is particularly high for OLLVM-b, OLLVM-bf, OLLVM-f, and OLLVM-s, amounting to at least 20%. The precise numbers are depicted in the confusion matrix in Figure 5.

While even more samples from the class Tigress-f are mistakenly predicted as no-obfuscation when using fewer features, it can still classify Tigress-o, Tigress-s, and Tigress-v correctly more than 90% of the time. It appears that the identifi-

cation of these techniques does not rely on the additional information provided by the omitted features as much as the identification of other methods.

6 Discussion

From the results, some interesting insights regarding obfuscation stealth can be gained, which we discuss in this section.

6.1 Stealthiness of obfuscations

The results show that the evaluated obfuscations have very different levels of stealthiness with respect to our model. The Tigress obfuscations virtualization (1.0), self-modifying code (1.0), and opaque predicates (0.96) could be identified best when making use of all features. The least identifiable protection classes were the Obfuscator LLVM classes OLLVM-fs (combination of flattening and self-modifying code, 0.45) and OLLVM-f (flattening, 0.73). As already pointed out in Section 5.3, this mainly results from the fact that the combinations of obfuscations are often confused with the standalone classes of the obfuscations they contain. However, significant differences can also be seen here. While LLVM-s and LLVM-fs can be distinguished from each other very well (0.02 and 0.0), in contrast, LLVM-f and LLVM-fs are often confused (0.25 and 0.47). This can be explained by the characteristics of the two obfuscations, instruction substitution and CFG flattening. CFG flattening modifies the structure of a program and, thus, its complexity significantly more than instruction substitution. For example, the cyclomatic complexity is directly affected by CFG flattening, whereas instruction substitution has no impact on it. This makes it easier to distinguish the combination of the two obfuscations from the one with less impact on code complexity.

It is also noteworthy that misclassifications are almost exclusively made within the classes of an obfuscator. The algorithmically very similar obfuscations LLVM-b and Tigress-o (bogus control flow using opaque predicates) and LLVM-f and Tigress-f (CFG flattening) are never or very rarely confused.

In Figure 4, the details of all obfuscations can be retrieved from the normalized confusion matrix.

6.2 Impact of compiler optimization levels

By extending the 11 obfuscation classes in our model by the compiler optimization levels O0 to O3 (i.e., a total of 44 instead of 11 obfuscation classes), the model's overall accuracy drops significantly to about 59%. Note that the hyperparameter tuning was performed for the pooled classes, though. The confusion matrix shows that the misclassifications occur mainly between classes with the same technique but different optimization levels. The approach including the optimization levels, thus significantly complicates the detection of the obfuscation technique in that model. In practice, however, the detection of the optimization

level has no relevance, which allowed us to combine the obfuscation classes in our model.

We also analyzed within which optimization levels the detection of obfuscations works best. We independently tested our model with only the programs from each of the four optimization levels (O0 to O3). While optimization level O0 is mainly used for debugging a program, the most frequently used optimization level for finished code is O2. In level O2, which is most commonly used in real-world programs, our model achieves an identification accuracy of 87.6%, while in level O0, it performs slightly better, reaching 89%.

Making a distinction between the optimization levels for the model prediction also provides insights into the level of stealthiness of each technique. For example, applying OLLVM-b with optimization level O0 makes it most challenging for the model to classify the obfuscation method correctly. We also observe an increase in obfuscation stealth when using optimization level O2 for Tigress-f as well as level O3 for Tigress-o.

7 Conclusions

In this paper, we have presented a novel methodology for identifying obfuscation techniques applied to code with high accuracy using a combination of static code complexity metrics. Unlike previous approaches, our methodology is not based on a direct analysis of the program code. It is thus more robust to semantic methods for increasing obfuscation stealth (e.g., normalization of opcode distribution). An evaluation with 11 obfuscation techniques and combinations of obfuscations in a total of 4124 programs has shown that the correct obfuscation technique can be predicted with an accuracy of 86.5%. Moreover, based on the classification results, we concluded that while tree-based methods are well suited for predicting obfuscation methods using code complexity, a well-trained neural network classifier can achieve even better results. Therefore, we chose the `MLPClassifier` as our best model, which can predict the underlying obfuscation method with an average accuracy of 83.9%. In future work, generating more data to fit the models could further improve predictability.

Our model can be applied within the scope of malware analysis. Due to a large number of new malware samples daily, efficient methods for obfuscation identification are needed to select suitable deobfuscation concepts.

Acknowledgments

This research was funded in whole, or in part, by the Austrian Science Fund (FWF) I 3646-N31.

References

1. Abran, A., Lopez, M., Habra, N.: An analysis of the mccabe cyclomatic complexity number. In: Proceedings of the 14th International Workshop on Software Measurement (IWSM) IWSM-Metrikon. pp. 391–405 (2004)
2. Bacci, A., Bartoli, A., Martinelli, F., Medvet, E., Mercaldo, F.: Detection of obfuscation techniques in android applications. In: Proceedings of the 13th International Conference on Availability, Reliability and Security. pp. 1–9 (2018)
3. Basili, V.R., Perricone, B.T.: Software errors and complexity: an empirical investigation0. Communications of the ACM **27**(1), 42–52 (1984)
4. Brosch, T., Morgenstern, M.: Runtime packers: The hidden problem. Black Hat USA (2006)
5. Canavese, D., Regano, L., Basile, C., Viticchié, A.: Estimating software obfuscation potency with artificial neural networks. In: Security and Trust Management: 13th International Workshop, STM 2017, Oslo, Norway, September 14–15, 2017, Proceedings 13. pp. 193–202. Springer (2017)
6. Coleman, D., Oman, P., Ash, D., Lowther, B.: Using metrics to evaluate software system maintainability. Computer **27**(08), 44–49 (08 1994)
7. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Tech. rep., Department of Computer Science, The University of Auckland, New Zealand (1997)
8. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 184–196 (1998)
9. Ebad, S.A., Darem, A.A., Abawajy, J.H.: Measuring software obfuscation quality–a systematic literature review. IEEE Access **9**, 99024–99038 (2021)
10. Ebert, C., Cain, J., Antoniol, G., Counsell, S., Laplante, P.: Cyclomatic complexity. IEEE software **33**(6), 27–29 (2016)
11. Fitzpatrick, J.: Applying the abc metric to c, c++, and java. Tech. rep., C++ Report (06 1997)
12. Gibert, D., Mateu, C., Planes, J., Vicens, R.: Classification of malware by using structural entropy on convolutional neural networks. Proceedings of the AAAI Conference on Artificial Intelligence **32**(1) (Apr 2018). https://doi.org/10.1609/aaai.v32i1.11409, https://ojs.aaai.org/index.php/AAAI/article/view/11409
13. Halstead, M.H.: Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., USA (1977)
14. Hatton, L.: Re-examining the defect-density versus component size distribution. IEEE Software p. 110 (1997)
15. Honglei, T., Wei, S., Yanan, Z.: The research on software metrics and software complexity metrics. In: 2009 International Forum on Computer Science-Technology and Applications. vol. 1, pp. 131–136. IEEE (2009)
16. Ikerionwu, C.: Cyclomatic complexity as a software metric. International Journal of Academic Research **2**(3) (2010)
17. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-llvm–software protection for the masses. In: 2015 ieee/acm 1st international workshop on software protection. pp. 3–9. IEEE (2015)
18. Kanzaki, Y., Monden, A., Collberg, C.: Code artificiality: a metric for the code stealth based on an n-gram model. In: 2015 IEEE/ACM 1st International Workshop on Software Protection. pp. 31–37. IEEE (2015)

19. Khan, A.A., Mahmood, A., Amralla, S.M., Mirza, T.H.: Comparison of software complexity metrics. International Journal of Computing and Network Technology **4**(01) (2016)
20. Kim, J., Kang, S., Cho, E.S., Paik, J.Y.: Lom: Lightweight classifier for obfuscation methods. In: Information Security Applications: 22nd International Conference, WISA 2021, Jeju Island, South Korea, August 11–13, 2021, Revised Selected Papers 22. pp. 3–15. Springer (2021)
21. Kurtukova, A., Romanov, A., Shelupanov, A.: Source code authorship identification using deep neural networks. Symmetry **12**(12) (2020)
22. Madi, A., Zein, O.K., Kadry, S.: On the improvement of cyclomatic complexity metric. International Journal of Software Engineering and Its Applications **7**(2), 67–82 (2013)
23. Madou, M., Anckaert, B., De Bus, B., De Bosschere, K., Cappaert, J., Preneel, B.: On the effectiveness of source code transformations for binary obfuscation. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP06). pp. 527–533. CSREA Press (2006)
24. Mason, J., Small, S., Monrose, F., MacManus, G.: English shellcode. In: Proceedings of the 16th ACM conference on Computer and communications security. pp. 524–533 (2009)
25. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering (4), 308–320 (1976)
26. McCabe, T.: A complexity measure. IEEE Transactions on Software Engineering **SE-2**(4), 308–320 (1976). https://doi.org/10.1109/TSE.1976.233837, https://doi.org/10.1109/TSE.1976.233837
27. Morgenstern, M., Pilz, H.: Useful and useless statistics about viruses and anti-virus programs. In: Proceedings of the CARO Workshop (2010)
28. Myers, G.J.: An extension to the cyclomatic measure of program complexity. SIGPLAN Not. **12**(10), 61–64 (10 1977)
29. Nagra, J., Collberg, C.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education (2009)
30. Necula, G.C., McPeak, S., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs (2002)
31. Oman, P., Hagemeister, J.: Metrics for assessing a software system's maintainability. In: Proceedings Conference on Software Maintenance 1992. pp. 337–344 (1992)
32. Oman, P., Hagemeister, J.: Construction and testing of polynomials predicting software maintainability. Journal of Systems and Software **24**(3), 251–266 (1994), oregon Workshop on Software Metrics
33. Rahbarinia, B., Balduzzi, M., Perdisci, R.: Exploring the long tail of (malicious) software downloads. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 391–402. IEEE (2017)
34. Sarwar, M.M.S., Shahzad, S., Ahmad, I.: Cyclomatic complexity: The nesting problem. In: Eighth International Conference on Digital Information Management (ICDIM 2013). pp. 274–279. IEEE (2013)
35. Sebastian, S.A., Malgaonkar, S., Shah, P., Kapoor, M., Parekhji, T.: A study & review on code obfuscation. In: 2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare. pp. 1–6. IEEE (2016)
36. Sellers, B.H.: Modularization and mccabe's cyclomatic complexity. Communications of the ACM **35**(12), 17–20 (1992)

37. Shen, V.Y., Yu, T.j., Thebaut, S.M., Paulsen, L.R.: Identifying error-prone software—an empirical study. IEEE Transactions on Software Engineering (4), 317–324 (1985)
38. Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. Advances in neural information processing systems **25** (2012)
39. Wang, Y., Rountev, A.: Who changed you? obfuscator identification for android. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). pp. 154–164. IEEE (2017)
40. Withrow, C.: Error density and size in ada software. IEEE Software **7**(1), 26–30 (1990)
41. Wu, Z., Gianvecchio, S., Xie, M., Wang, H.: Mimimorphism: A new approach to binary code obfuscation. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 536–546 (2010)
42. Yu, S., Zhou, S.: A survey on metric of software complexity. In: 2010 2nd IEEE International conference on information management and engineering. pp. 352–356. IEEE (2010)

A Specifications

Table 4. Best parameter combinations found per classifier and feature set.

	All features	**5/14 features**
Extra Trees	n_estimators = 80 min_samples_split = 3 min_samples_leaf = 1 max_features = None max_depth = 33 criterion = 'gini' bootstrap: 'False'	n_estimators = 200 min_samples_split = 2 min_samples_leaf = 1 max_features = None max_depth = 24 criterion = 'entropy' bootstrap: 'False'
LGBM	subsample = 1 objective = 'multiclass' num_leaves = 30 max_depth = -1 learning_rate = 0.2 colsample_bytree = 1 boosting_type = 'gbdt'	subsample = 0.5 objective = 'multiclass' num_leaves = 250 max_depth = 30 learning_rate = 0.10923689091995176 colsample_bytree = 1 boosting_type = 'gbdt
Random Forest	n_estimators = 80 min_samples split = 2 samples_leaf = 1 max_features = 'sqrt' max_depth = 50 criterion = 'entropy' bootstrap: 'False'	n_estimators = 164 min_samples split = 3 min_samples leaf = 1 max_features = 'sqrt' max_depth = 27 criterion = 'entropy' bootstrap: 'False'
MLP	hidden_layer_sizes = (50, 100) activation = 'tanh' solver = 'lbfgs' batch_size = 64 learning_rate = 'invscaling' alpha = 0.08745094461679037 learning_rate_init = 0.0001 early_stopping = True	hidden_layer_sizes = (75, 100) activation = 'tanh' solver = 'lbfgs' batch_size = 16 learning_rate = 'invscaling' alpha = 0.1 learning_rate_init = 0.0033083971654978557 early_stopping = True