Journal of Information Security and Applications 85 (2024) 103850

Contents lists available at ScienceDirect

Journal of Information Security and Applications

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

Obfuscation undercover: Unraveling the impact of obfuscation layering on structural code patterns

Sebastian Raubitzek [b], Sebastian Schrittwieser [a,*], Elisabeth Wimmer [b], Kevin Mallinger [a]

[a] CD Laboratory for Assurance and Transparency in Software Protection, Faculty of Computer Science, University of Vienna, Austria
[b] SBA Research, Vienna, Austria

ARTICLE INFO

Keywords:
Software obfuscation
Code structure
Obfuscation layering
Classification

ABSTRACT

Malware often uses code obfuscation to evade detection, employing techniques such as packing, virtualization, and data encoding or encryption. Despite widespread application, the impact of combining these techniques in a particular order – so-called obfuscation layering – on code analysis remains poorly understood. This study advances previous research by examining the effects of obfuscation layering on the classification of obfuscation techniques contained in binary code, focusing on how different layering combinations alter structural code patterns. Utilizing a dataset of 85 C programs modified with various combinations of code obfuscation techniques, we analyze the impact of obfuscation layering on structural code metrics such as its control flow complexity. Our study demonstrates that obfuscation layering significantly affects the ability to classify obfuscated code and that the order of applied obfuscations is less significant for classification than previously assumed. Through explainability methodologies our work offers novel insights for malware analysts and researchers to improve their detection strategies.

1. Introduction

Malware authors can choose from a variety of different obfuscation techniques to make their code more difficult to detect and analyze. And indeed, code obfuscation is heavily used in practice. A Black Hat survey by [1] 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, [2] 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 [3]. Beyond packing, other obfuscation methods such as virtualization, data encoding/encryption, and concealing libraries are also widely adopted by malware authors.

As opposed to the goal of malware authors to hide the maliciousness of their code, the aim of malware analysts and researchers is to efficiently analyze unknown malware samples and understand their functionality. A critical aspect of analyzing obfuscated code is the identification of the obfuscation techniques employed. Knowing the methods of obfuscation used in a sample greatly speeds up the analysis process, as tailored de-obfuscation techniques are available for many obfuscations.

In this work, we extend our previous research [4] on code obfuscation classification based on modeling structural code patterns through an in-depth analysis of obfuscation layering, i.e., the combination of different protection techniques in a particular order. While there exists a broad consensus in the software protection community that only a combination of multiple protections can achieve an adequate level of security, obfuscation layering has so far received little attention in the literature and it is mostly unexplored what effects the layering of protections has on the resulting programs and to what extent the individual obfuscations can be detected through structural code patterns. The underlying idea of obfuscation classification through structural code patterns is that each obfuscation technique introduces characteristic modifications to the structure of the program code. For example, code flattening reduces a program's hierarchical complexity, which can be observed in a reduced depth of the control flow graph. At the same time, however, minimal changes to the distribution of opcodes are made, as, put simply, only jump targets are replaced without modifications to the opcode (e.g., an unconditional JMP instruction) itself. Another widely-used obfuscation technique is instruction substitution, which replaces individual instructions or groups of instructions with semantically equivalent ones. When looking at the code structure, the distribution of opcodes will change significantly, but the hierarchical complexity of the program does not change at all. In our obfuscation classification approach [4], we measure a set of code structure metrics

* Corresponding author.
E-mail addresses: sraubitzek2@sba-research.org (S. Raubitzek), sebastian.schrittwieser@univie.ac.at (S. Schrittwieser), ewimmer@sba-research.org (E. Wimmer), kevin.mallinger@univie.ac.at (K. Mallinger).

https://doi.org/10.1016/j.jisa.2024.103850

Available online 16 August 2024

and use the results to classify the obfuscation techniques applied to the samples. However, obfuscation layering was not analyzed in detail in our previous work. As obfuscation techniques that strongly modify the code structure of a program might cover earlier applied techniques that modify the same code structures, our hypothesis was that obfuscation layering significantly reduces the quality of our obfuscation classifier. The aim of this work therefore is to systematically evaluate which combinations of techniques have an impact on the classification and thus to gain a better understanding for obfuscation layering and the identification of individual techniques in protected code. The three aspects of our investigation can be summarized as follows:

1. To what extent are code (complexity) metrics useful for identifying obfuscations, particularly a layering thereof?
2. Which complexity metrics are the most expressive when it comes to identifying obfuscated code?
3. How does the layering of obfuscations affect the identification of the used obfuscations?

For this paper we studied the effects of obfuscation layering specifically on binary code, thus script languages such as JavaScript and byte code such as Java are out of scope of this work. Starting from a set of 85 C programs, we created a comprehensive research dataset based on the Tigress obfuscator introduced by [5]. We treated the 85 programs with 16 different Tigress configurations, each in four compiler optimization levels (resulting in 64 configurations). Together with 16 variants of non-obfuscated binaries using different compilers, we reached a total of 80 different build configurations. Since not all configurations lead to a valid binary program for each sample, we received a total of 6211 programs for the structural analysis of layered obfuscations. To the best of our knowledge, our work is the first in-depth analysis of the effects of obfuscation layering applied to a comprehensive dataset.

The remainder of this paper is structured as follows. Section 2 discusses related work, while Section 3 presents the fundamentals of code obfuscation and the measurement of structural code patterns. Section 4 introduces our methodology, and Section 5 presents the results and discusses them. Section 6 concludes the paper.

2. Related work

Previous literature described a protection technique as stealthy if the resulting code resembles the original code as much as possible [6]. One major problem with quantifying stealthiness of an obfuscation technique is that it highly depends on the structure of the original program whether or not the 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, e.g., in terms of code structure. [7] described two types of obfuscation stealth. Local stealth measures the difficulty of identifying the exact location of an obfuscation applied to code. In contrast, steganographic stealth describes the difficulty of detecting if a specific obfuscation was applied at all. Measuring obfuscation stealth is not trivial. 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 and 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 or other obfuscations. 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, past literature proposed approaches such as using Huffman encodings [8] to make the packed code look structurally like actual binary code or shell code that looks like English prose [9]. While English prose can clearly be distinguished from actual shell code, the context where shell code is utilized (e.g., as part of natural language text sent to a system) makes it a perfect camouflage. Thus, coverage alone is not a good indicator of the stealthiness of an obfuscation.

Previous approaches to obfuscation detection are mainly founded on basic code structures such as opcode frequencies. [10] 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, e.g., control-flow modifying obfuscations such as CFG flattening, have a minimal effect. [11] 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 [12] in 2018. It utilizes features extracted from the Smali representation of the application's bytecode. [13] proposed a machine-learning based approach for the detection of class-level obfuscations in Android applications. Another machine-learning based obfuscation identification technique which can be applied to binary code was introduced by [14]. A model for detecting bogus control flow transformations added with the Obfuscator-LLVM framework was trained using an annotation approach in which the obfuscator's transformations were first annotated and used to create labels for unannotated binaries. LOM by [15] uses a neural network-based classifier on the opcode distribution of binary code for obfuscation identification. [16] extracted Term Frequency Inverse Document Frequency (TF-IDF) features from Tigress-protected samples for the identification of six different obfuscation methods. [17] used a 5-gram birthmark of Java bytecode to identify the obfuscation tool used for protection. [18] introduced a trigram based classifier for the detection of boot sector viruses.

3. Preliminaries

3.1. Code obfuscation

Obfuscating transformations convert code – either in the form of source, byte, or executable code – into code that is more difficult to understand for a human code analyst and/or difficult to process for automated code analysis tools (e.g., the obfuscation might make a static disassembling tool fail on the binary). 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 aiming to hide the malicious purposes of their code. Thus, identifying obfuscations in binary code is a fundamental prerequisite of malware detection and analysis. [19] categorized code obfuscations 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).

Obfuscations are usually applied to code through automatic tools. Some commercial source code protection solutions are offered on the market (e.g., Cloakware by Irdeto[1]), but also many freely available tools and online services exist. In the scientific community, the Tigress obfuscator by [5] is widely used. Tigress is a C source-to-source obfuscator, meaning that the obfuscating transformations are applied to C source code and the protected code is returned as C source code. It was developed based on the C Intermediate Language CIL [20] and MyJit[2] and can protect code with a variety of obfuscation methods.

[1] https://irdeto.com (Accessed: March, 14th 2023)
[2] https://myjit.sourceforge.net (Accessed: March, 14th 2023)

Table 1
Applied obfuscations.

Technique	Abbreviation	Description
Opaque predicates	Opa	Makes it more difficult to evaluate expressions for conditional jumps
Virtualization	Virtualize	Transforms binary code to byte code of a custom virtual machine
CFG flattening	Flatten	Redirects all control-flow transfers to a central dispatcher
Mixed-Boolean arithmetic (MBA)	Encode	Replaces simple integer arithmetic with complex expressions
Function splitting	Split	Splits functions in two ore more smaller functions
Anti branch analysis	Opabaea	Makes it more difficult to determine the target of branches. Only used in combination with opaque predicates and MBA in our work.

However, considerable uncertainty exists as to whether – and, if so, to what extent – a protection is transferred to the binary program during compilation. [21] demonstrated empirically that not all types of protection survive the compilation process. This undesired effect results from the fact that software protections intentionally make code more complicated. A compiler, however, 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 [22], it was prototypically demonstrated that compile-time protection of code is feasible. However, implemented protections are of low complexity and tailored de-obfuscation algorithms exist.[3]

In this work, we use the Tigress obfuscator for sample generation. Specifically, we used six different classes of obfuscations and 16 layering combinations. Table 1 describes the applied obfuscations. In Appendix A, the Tigress configurations for all 16 combinations are listed for reproducibility.

3.2. Structural code patterns

Program code has a variety of structural patterns from which different program properties can be derived. The opcode distribution, for example, can be used to draw conclusions about its functionality. Previous literature has shown that cryptographic algorithms (e.g., in ransomware) can be identified by the high share of arithmetic and logical operations [23]. Other structural patterns of program code include cyclomatic complexity, the number of Boolean or logic tests in the program, its branching behavior, and the number of storage or transfer operations of data into a variable. Since the 1970s, several metrics have been presented in the literature that attempt to quantify these structural patterns in program code. In software development, these are often used to evaluate the complexity of a program, although not all of them were specifically designed for measuring code complexity. In the following, we present the metrics we used in our obfuscation classification methodology.

3.2.1. Halstead complexity metrics

As an early pioneer of software science, [24] was one of the first to analyze software and its structures quantitatively. His work resulted in the formalization of the *Halstead complexity metrics*, which consist of several sub-metrics.

[3] https://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html (Accessed: March, 14th 2023)

- *Halstead difficulty* measures how difficult it is to write or understand the code of a program. It is defined as $D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$, 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 $V = N \cdot \log_2 n$, where $N = N_1 + N_2$ (N_1 is the total number of operators) and $n = n_1 + n_2$.
- *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 is defined as $T = \frac{E}{18}$. Since Halstead time differs from the Halstead effort by a constant factor only, we excluded it from our measurements.

3.2.2. Cyclomatic complexity

Cyclomatic Complexity describes the structure of software through the number of possible independent paths in its control flow graph [25–31]. To calculate McCabe's cyclomatic complexity, 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 is the number of nodes, and p is the number of connected entities in G. In code obfuscation, it was used in the past to measure the strength of protecting transformations [32].

Myer's interval [33] is an extension of McCabe's cyclomatic complexity. It is defined as $v(G) : v(G) + L$ adds the number of logical operators L to the measure.

3.2.3. ABC metric

Despite its traditional categorization as a size metric, the *ABC metric* [34] 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 $|ABC| > 0$.

3.2.4. Maintainability index

As the name already suggests, the *maintainability index* [35–37] 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 the lines of code (LoC). It is defined as $MIwoc = 171 - 5.2 \cdot \ln(HV) - 0.23 \cdot CC - 16.2 \cdot \ln(LOC)$.

3.2.5. Applications for code obfuscation

These code structure metrics were initially created to help build reliable, readable, and maintainable software constructs. A higher value indicates a more complex code structure with regards to the measured properties of a program. [38] first suggested to use code structure metrics for measuring obfuscation potency, i.e., how well humans are able to comprehend the code. Obfuscating transformations make code artificially more complicated, reducing its comprehensibility for humans. And indeed, a recent literature survey by [39] confirms that code structure metrics are frequently used to measure obfuscation potency. In 2023, we first showed how code structure metrics can also be used for measuring obfuscation stealth [4]. Obfuscations can be classified effectively by their characteristic modifications to the code structure.

4. Approach

For our study, we created a dataset from 85 programs which we treated with various obfuscation configurations using the Tigress obfuscator. These programs are single-function and single-file C source codes from different categories, such as hashing and sorting algorithms. In addition to self-written programs, we included samples from the obfuscation benchmark repository by [40], which was composed with the aim of representing both typical algorithms and varying code characteristics (e.g., number of loops, depth of nested control flow, etc.). All 85 programs were then compiled and obfuscated in different combinations. First, a set of non-obfuscated samples was generated using four different compilers (GCC, clang, Tendra, and TinyCC). For GCC and clang we created samples on four different compiler optimization levels (O0 to O3) each. Then, we applied a total of 16 different obfuscation configurations to the 85 programs using Tigress. The focus was put on the layering of obfuscations, i.e., the sequential application of two or more protections to the program code to explore how layering affects the structural properties of the generated programs. Starting from the hypothesis that the order of the application of different obfuscations significantly impacts the structural properties of the generated samples, the goal was to determine to what extent the structural properties of earlier applied obfuscations are obscured by later obfuscations. For achieving this goal, we discuss the classification performances of several ML classifiers for which we give as inputs the structural code patterns of an obfuscated program and ask to guess which obfuscation techniques have been applied to the original program.

We did not create all possible layering combinations but instead selected a subset that includes layerings of protections affecting similar structural properties of the code and those working on different properties. Special focus was placed on the last protection applied, as we hypothesized that it has the most significant impact on the final structure of the program.

This first step of the pipeline is depicted in the left part of Fig. 1, and denoted as the dataset part.

4.1. Data preparation

We pooled the classes corresponding to the Tigress configuration, i.e., we combined the different optimization levels per obfuscation class. Further, we created a single class for all non-obfuscated samples from the different compilers. In total, we ended with 17 classes (one non-obfuscated and 16 obfuscation classes). The Tigress configurations are listed in Appendix A. Not all obfuscating treatments of the original 85 programs resulted in valid binaries. We performed a functional check for each binary program and automatically discarded broken ones.

The code structure metrics used for classification were originally developed for source code and have limited applicability for the measurement of the complexity of binary code. We therefore disassembled each binary sample using the Rizin disassembler toolkit[4] to obtain an assembly representation. Disassembling binary code is an error-prone process, especially when obfuscation techniques have been applied to the code. Thus, there is a risk that the disassembled code is not completely error-free and subsequently the calculation of the code structure measures might also be slightly incorrect, which in turn could have a negative impact on the classification capabilities of the model. However, since we do not make any statements about the actual complexity of code, but rather use the measures as features for our machine learning methodology, we do not consider potential data imprecision to be critical. Some metrics can be applied directly to assembly code. In the Halstead measures, for example, the assembly operators (opcodes) and their operands are used directly. For other metrics, we have designed our own mapping for assembly code. For the ABC metric, for example, we created a set of data manipulating instructions, including arithmetic and logical operations, shift rotates, etc. for the assignment count, CALL, JMP, RET, etc. instructions for the branch count, and conditionals (e.g., a conditional jump or call) for the condition count.

It is clear from the correlation matrix in Fig. 2 that some of the features are highly correlated. This is due to the nature of the metrics. Some of them are combinations or extensions of each other, as described in Section 3.2. For example, the ABC metric is composed of A, B, and C, which are also included in the original feature set. Nonetheless, given our specific goal of understanding how structural code patterns influence model effectiveness, we maintained all complexity metrics as features. This approach ensures an exhaustive evaluation by analyzing the full range of complexity metrics, allowing for a more detailed assessment of the impact of each structural pattern on the model's performance.

4.2. Experimental setup

For our training and testing setup, we conducted two separate trials. In the first trial, we implemented a randomized 80/20 split for the training and testing data. This involved shuffling all samples and allocating 20% for testing while the remainder was used for training. In the second trial, we divided the training and testing datasets based on programs. Out of 85 programs, we selected 68 for training and 17 for testing, which again corresponds to an approximate 80/20 split. This strategy allowed us to evaluate concepts of training and testing splits, enabling us to determine whether our approach is agnostic to the presence of the same program in both the training and testing sets. In other words, we assessed whether a program is exclusively in either the training or testing set to estimate if our approach is generalizable. Additionally, to address the issue of slightly imbalanced data (our dominant class is non-obfuscated code), we employed the ADASYN (Adaptive Synthetic Sampling, [41]) technique to generate synthetic data, thereby balancing the dataset and increasing the number of samples per obfuscation class to 2000. We employed ADASYN for both the 80/20 split and the program-wise split and always after the train/test split and only on the training dataset. We rescaled all data before training using Sci-kit Learn's `StandardScaler` [42], i.e., shifting the data to have a mean of zero and a standard deviation of 1.

We decided for models that are both explainable and interpretable to potentially identify patterns for classifying obfuscation classes. This approach leverages well-established strategies for interpreting machine learning models, notably through the use of SHAP (SHapley Additive exPlanations) values [43] and feature importance analysis [44]. These techniques provide insights into the contribution of each feature to the model's predictions, enabling a deeper understanding of the underlying decision-making processes. From a data perspective, they offer interpretable insights into the data. SHAP values [43], in particular, quantify

[4] https://rizin.re (Accessed: March, 14th 2024)

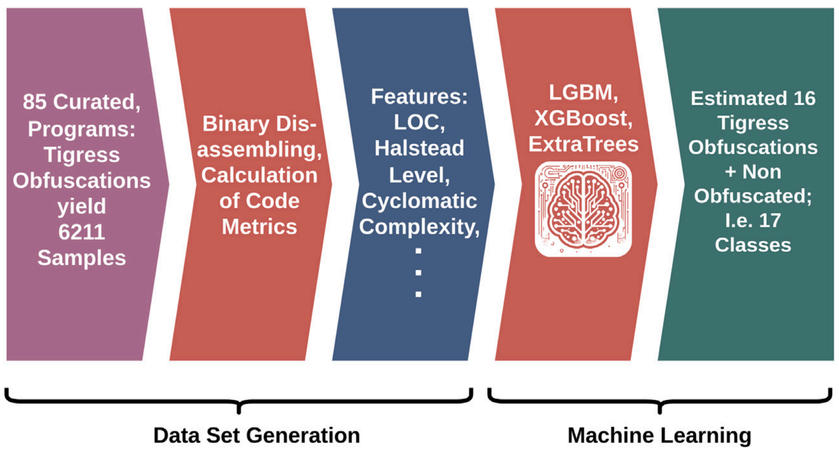

85 Curated, Programs: Tigress Obfuscations yield 6211 Samples
Binary Dis-assembling, Calculation of Code Metrics
Features: LOC, Halstead Level, Cyclomatic Complexity, . . .
LGBM, XGBoost, ExtraTrees
Estimated 16 Tigress Obfuscations + Non Obfuscated; I.e. 17 Classes
Data Set Generation
Machine Learning

Fig. 1. Schematic depiction of our pipeline. The left part illustrates the constituents of the dataset, i.e., the samples generated from the 85 source code files and the information on their obfuscation, transformed into a dataset with structural code metrics. The right part depicts our machine learning approach, highlighting the three employed classifiers, and the estimated obfuscation type.

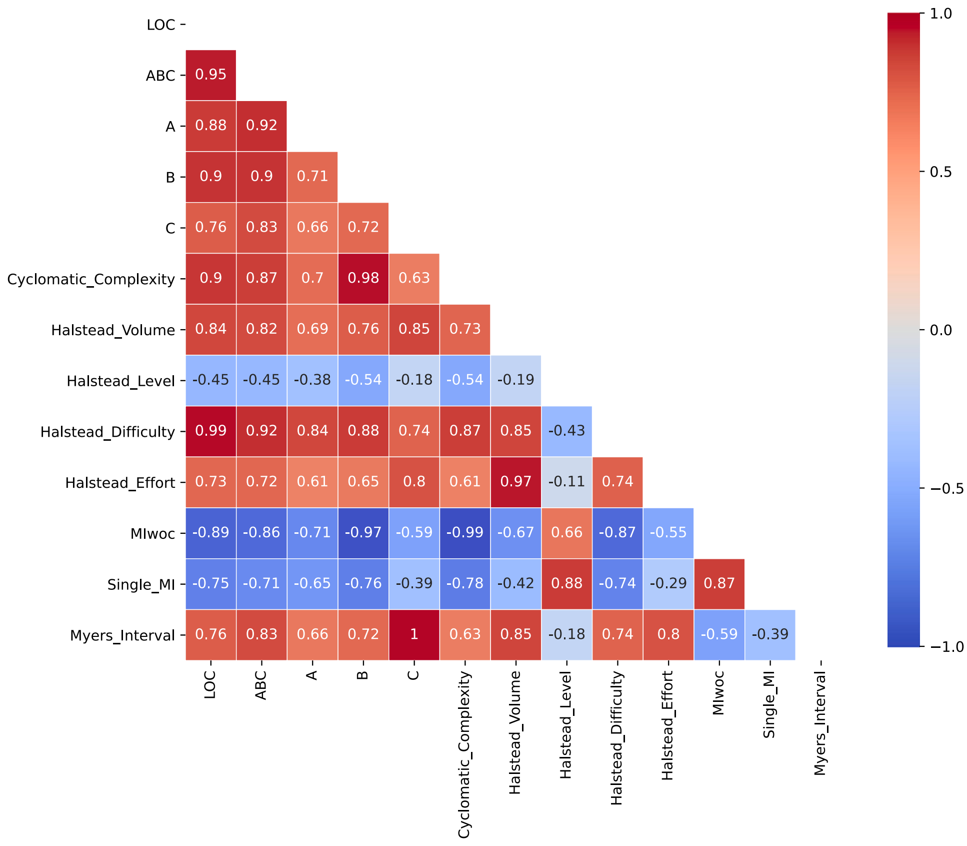


Fig. 2. Correlation coefficients for all code structure metrics.

the impact of each feature on the prediction outcome. Incorporating feature importance analysis, especially for gradient boosting classifiers, enhances our model's interpretability by identifying the most influential attributes in predicting obfuscation schemes. This methodology is consistent with established practices in tree-based modeling, as outlined in the scikit-learn documentation, [44].

For our analysis, we thus selected Extremely Randomized Trees (ExtraTrees, [45]), Extreme Gradient Boosting (XGBoost, [46]), and Light Gradient Boosting Machine (LightGBM, [47]) as our classifiers, as we can make use of the previously outlined interpretation techniques, as depicted in Fig. 1. Further, these methods are known for their high performance across various problems [48–51], and were indicated to perform well by employing lazy learner, [52].

To optimize the hyperparameters of each model, we conducted a 100-step Bayesian optimization with 5-fold cross-validation on extensive parameter grids for each algorithm to find the optimal model, i.e., the optimal hyperparameters. Hyperparameter tuning via Bayesian optimization is a model-based optimization algorithm that leverages past loss data to determine an optimal parameter set for each model. Using prior information in Bayesian optimization significantly increases its efficiency compared to traditional methods such as grid search or random search, as highlighted by [53]. For this hyperparameter

Table 2
Algorithms Performances for the Test dataset. 80/20 Split.

	Accuracy	Recall	Precision	F1 score
Extra trees	0.5712	0.5876	0.5712	0.5783
LGBM	**0.5784**	**0.5960**	**0.5784**	**0.5862**
XGBoost	0.5632	0.5813	0.5632	0.5715

Table 3
Algorithms Performances for the Test dataset. Program-wise 68/17 Split.

	Accuracy	Recall	Precision	F1 score
Extra trees	**0.6869**	**0.6781**	**0.6869**	**0.6795**
LGBM	0.6645	0.6653	0.6645	0.6631
XGBoost	0.6669	0.6661	0.6669	0.6638

optimization, we used an existing implementation from `Scikit-optimize`. The accuracy of each model was then evaluated using a range of metrics, including classification accuracy, F1 score, precision, and recall. The full code and data can be found in our corresponding GitHubRepository.

5. Results

In this section, we discuss our best models, and the achieved results. Specifically, we detail the outcomes of our experiments, classification metrics, and corresponding confusion matrices. Additionally, we study the interpretation of our models through SHAP analysis and discuss the feature importance of each model. Given the extensive nature of the results, our focus will be on the classifiers that achieved the highest accuracy, with supplementary results provided in Appendix B.

5.1. Model results

As indicated in Tables 2 and 3, which show the results for our test data, we see that for the regular 80/20 split, the ExtraTrees Classifier performed best, whereas for the program-wise split, the LGBM classifier performed best. Here it is surprising that the results for the program-wise split are significantly better than the ones for the regular split. Meaning that our best results are approximately ≈ 0.1 better for the program-wise split. I.e., the ExtraTrees classifier trained on program-wise-split data achieved an accuracy of approximately 0.6869, whereas the best LGBM classifier trained on the 80/20-split data achieved an accuracy of approximately 0.5784. The same is true for all other employed metrics featured in Tables 2 and 3. This increase in performance for the program-wise split is also depicted in our cross-validation results presented in Table 4, i.e., the results for the program-wise split always outperform the results for the 80/20 split. This table shows the best cross-validation accuracy scores for both train–test-split scenarios and all algorithms. These are the best results achieved by the Bayesian Cross Validation Search, meaning that these scores depict how well each algorithm performs on the training dataset.

Thus, we see a significant impact of the train–test split on machine learning model performance, which is markedly evident in our study for both train and test data, i.e., when comparing cross-validation (CV) scores and test scores. This significant variance underscores the impact of the split choice on training effectiveness and model validation. Importantly, the program-wise split, which ensures all versions of one obfuscated program are contained within the same subset, appears to provide a more comprehensive set of data, suggesting that a program-wise split enhances generalizability, as not only the accuracy of test data is improved, but also the accuracy on training data.

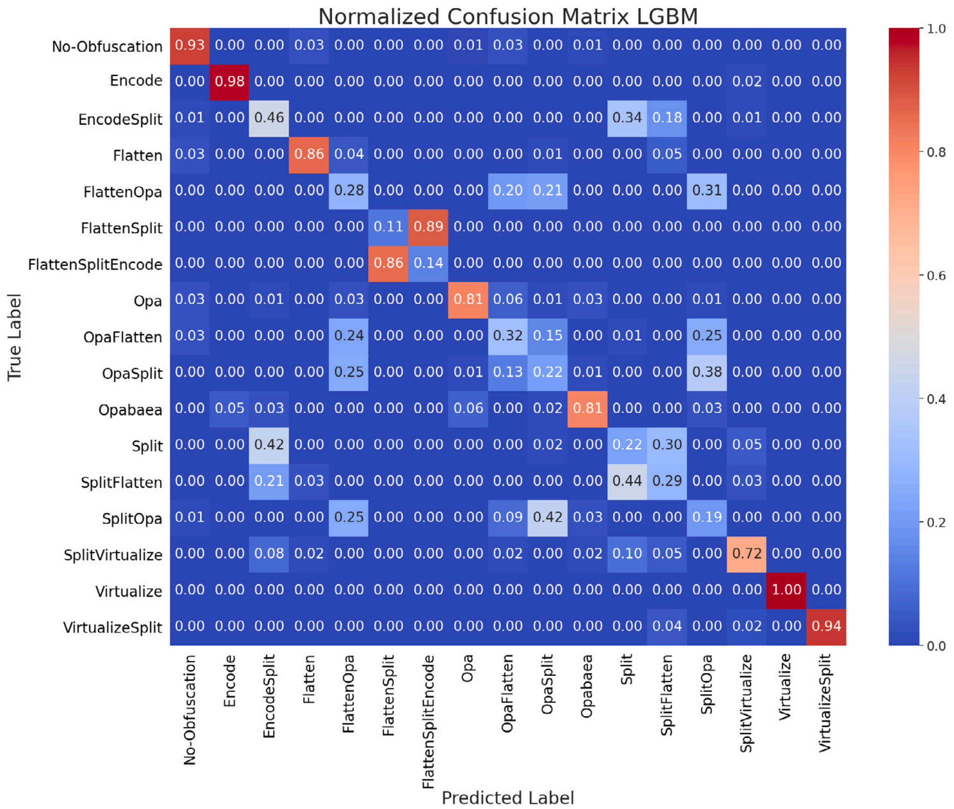
We also depicted these results in confusion matrices 3 and 4 for the two best-performing models, thus highlighting the results on individual classes for the test data. The remaining confusion matrices are collected in Appendix B.1. The featured confusion matrices show for both splits that we have classes that can be classified with a very

Table 4
Best CV Accuracy Scores for Each Algorithm Across Different Splits. This score is the accuracy obtained for the best model during hyperparameter optimization.

Algorithm	80/20 Split	Program-wise 68/17 Split
Extra trees	0.7591	**0.7957**
LGBM	**0.7705**	0.7802
XGBoost	0.7574	0.7671

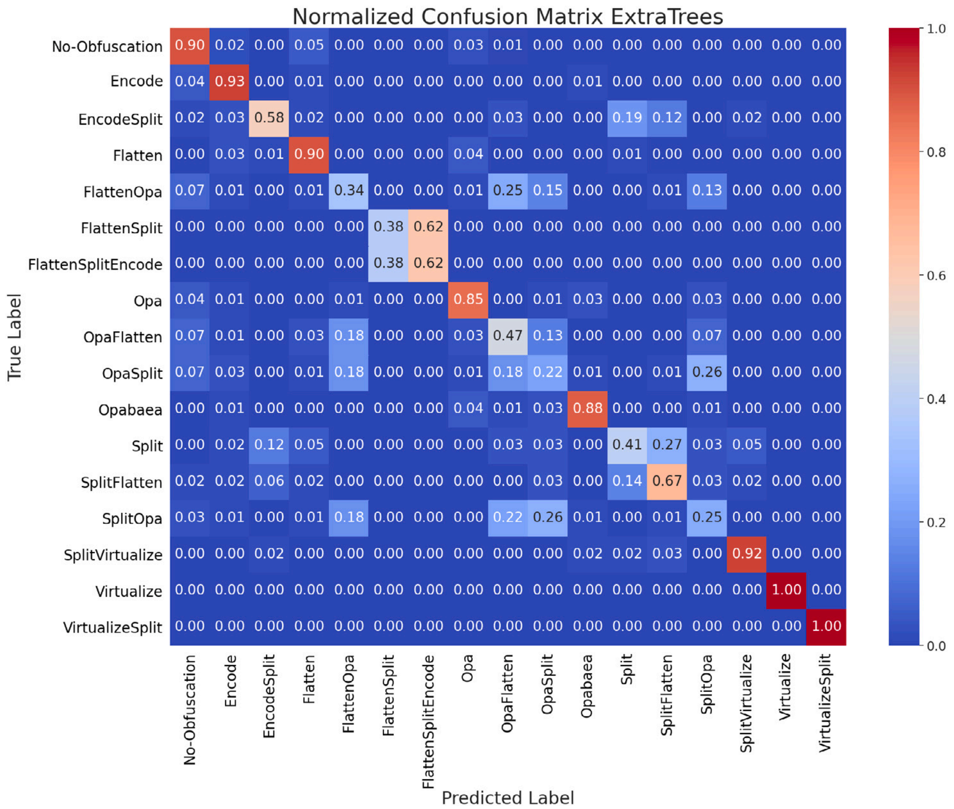
high accuracy, i.e., 0.9 and above, some even with 1.0, highlighting a perfect classification of these classes in the training dataset. For all confusion matrices, we see that classes the classes `No-Obfuscation`, `Encode`, `Virtualize` and `VirtualizeSplit` are classified with high accuracy, i.e., 0.9 and above. Here, `Virtualize` and `VirtualizeSplit` are oftentimes classified with perfect accuracy, i.e., no single one of these instances was classified wrong. This not only shows the strength of our approach in, first of all, differentiating between obfuscated and non-obfuscated code but also shows that code treated with certain obfuscation combinations is particularly vulnerable to being identified as originating from this obfuscation configuration. In contrast, other obfuscations and or layered obfuscations do not allow to being identified that easily. Here, we want to emphasize three particular clusters that are present in all of our results. I.e., obfuscations that are often mistaken for each other and/or not classified correctly. The first pair here is the `FlattenSplit` and `FlattenSplitEncode`; these two combinations are particularly strongly mistaken for each other, as e.g. Confusion Matrix Fig. B.9 shows where `FlattenSplit` is identified as `FlattenSplitEncode` on average with a score of 0.86. The other two clusters that exhibit similar behavior for being mistaken are `OpaFlatten` and `OpaSplit`; and finally `Split` and `Flatten`.

Focusing on the obfuscation layering's impact on classification, our analysis revealed significant insights into how certain obfuscations obscure others' structural patterns when layered together. This intricacy is, as previously mentioned, particularly evident in the confusion across obfuscation techniques that share at least one layer, e.g. `_Opa*`, `_Flatten*`, and so on. Our initial hypothesis was that the order of applied code transformations plays a critical role in the structure of the resulting binary code. Obfuscations applied later would override structural changes made by earlier obfuscations if they modify similar structural code properties. This would lead to confusions between classes that share the same transformations at the end of the layering process. In two of the three identified clusters with particularly high confusions, we observed that transformations applied earlier in the layering process are responsible for the observed confusions. In the cluster `FlattenSplit` and `FlattenSplitEncode`, the only difference in the layering is the `Encode` transformation. This transformation modifies data structurally, unlike `Flatten` and `Split`, which alter the control flow graphs of the programs. Therefore, `Encode` is unable to hide the characteristic properties of the `Flatten` and `Split` transformations, resulting in increased confusions between the two layerings. In the cluster `OpaFlatten` and `OpaSplit`, the last transformations, `Flatten` and `Split`, modify similar structural code properties (i.e., the control flow graphs), and thus, the confusions seem explainable here. However, it must be assumed that the initially applied `Opa` transformation in both classes significantly influences the observed confusions, as such confusions do not occur in other classes with `Split` and `Flatten` at the end of the layering process (e.g., `EncodeSplit` and `OpaFlatten`). Furthermore, in the classes `EncodeSplit` and `SplitFlatten`, slightly increased confusions can be observed (0.12). We conclude that the influence of covering transformations through later applied ones is less significant than initially assumed, and the contained transformations have a bigger impact on classification accuracy. The classifier often struggled to distinguish between classes with a shared subset of transformations, thus highlighting a challenge in identifying layered obfuscations correctly.



True Label \ Predicted Label	No-Obfuscation	Encode	EncodeSplit	Flatten	FlattenOpa	FlattenSplit	FlattenSplitEncode	Opa	OpaFlatten	OpaSplit	Opabaea	Split	SplitFlatten	SplitOpa	SplitVirtualize	Virtualize	VirtualizeSplit
No-Obfuscation	0.93	0.00	0.00	0.03	0.00	0.00	0.00	0.01	0.03	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
Encode	0.00	0.98	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00
EncodeSplit	0.01	0.00	0.46	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.34	0.18	0.00	0.01	0.00	0.00
Flatten	0.03	0.00	0.00	0.86	0.04	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.05	0.00	0.00	0.00	0.00
FlattenOpa	0.00	0.00	0.00	0.00	0.28	0.00	0.00	0.00	0.20	0.21	0.00	0.00	0.00	0.31	0.00	0.00	0.00
FlattenSplit	0.00	0.00	0.00	0.00	0.00	0.11	0.89	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
FlattenSplitEncode	0.00	0.00	0.00	0.00	0.00	0.86	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Opa	0.03	0.00	0.01	0.00	0.03	0.00	0.00	0.81	0.06	0.01	0.03	0.00	0.00	0.01	0.00	0.00	0.00
OpaFlatten	0.03	0.00	0.00	0.00	0.24	0.00	0.00	0.00	0.32	0.15	0.00	0.01	0.00	0.25	0.00	0.00	0.00
OpaSplit	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.01	0.13	0.22	0.01	0.00	0.00	0.38	0.00	0.00	0.00
Opabaea	0.00	0.05	0.03	0.00	0.00	0.00	0.00	0.06	0.00	0.02	0.81	0.00	0.00	0.03	0.00	0.00	0.00
Split	0.00	0.00	0.42	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.22	0.30	0.00	0.05	0.00	0.00
SplitFlatten	0.00	0.00	0.21	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.44	0.29	0.00	0.03	0.00	0.00
SplitOpa	0.01	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.09	0.42	0.03	0.00	0.00	0.19	0.00	0.00	0.00
SplitVirtualize	0.00	0.00	0.08	0.02	0.00	0.00	0.00	0.00	0.02	0.00	0.02	0.10	0.05	0.00	0.72	0.00	0.00
Virtualize	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
VirtualizeSplit	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.02	0.00	0.94

Fig. 3. Confusion matrix for LGBM, 80/20 train test split.



True Label \ Predicted Label	No-Obfuscation	Encode	EncodeSplit	Flatten	FlattenOpa	FlattenSplit	FlattenSplitEncode	Opa	OpaFlatten	OpaSplit	Opabaea	Split	SplitFlatten	SplitOpa	SplitVirtualize	Virtualize	VirtualizeSplit
No-Obfuscation	0.90	0.02	0.00	0.05	0.00	0.00	0.00	0.03	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Encode	0.04	0.93	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
EncodeSplit	0.02	0.03	0.58	0.02	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.19	0.12	0.00	0.02	0.00	0.00
Flatten	0.00	0.03	0.01	0.90	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00
FlattenOpa	0.07	0.01	0.00	0.01	0.34	0.00	0.00	0.01	0.25	0.15	0.00	0.00	0.01	0.13	0.00	0.00	0.00
FlattenSplit	0.00	0.00	0.00	0.00	0.00	0.38	0.62	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
FlattenSplitEncode	0.00	0.00	0.00	0.00	0.00	0.38	0.62	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Opa	0.04	0.01	0.00	0.00	0.01	0.00	0.00	0.85	0.00	0.01	0.03	0.00	0.00	0.03	0.00	0.00	0.00
OpaFlatten	0.07	0.01	0.00	0.03	0.18	0.00	0.00	0.03	0.47	0.13	0.00	0.00	0.00	0.07	0.00	0.00	0.00
OpaSplit	0.07	0.03	0.00	0.01	0.18	0.00	0.00	0.01	0.18	0.22	0.01	0.00	0.01	0.26	0.00	0.00	0.00
Opabaea	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.04	0.01	0.03	0.88	0.00	0.00	0.01	0.00	0.00	0.00
Split	0.00	0.02	0.12	0.05	0.00	0.00	0.00	0.00	0.03	0.03	0.00	0.41	0.27	0.03	0.05	0.00	0.00
SplitFlatten	0.02	0.02	0.06	0.02	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.14	0.67	0.03	0.02	0.00	0.00
SplitOpa	0.03	0.01	0.00	0.01	0.18	0.00	0.00	0.00	0.22	0.26	0.01	0.00	0.01	0.25	0.00	0.00	0.00
SplitVirtualize	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.03	0.00	0.92	0.00	0.00
Virtualize	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
VirtualizeSplit	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Fig. 4. Confusion matrix for ExtraTrees, 68/17 program-wise split.

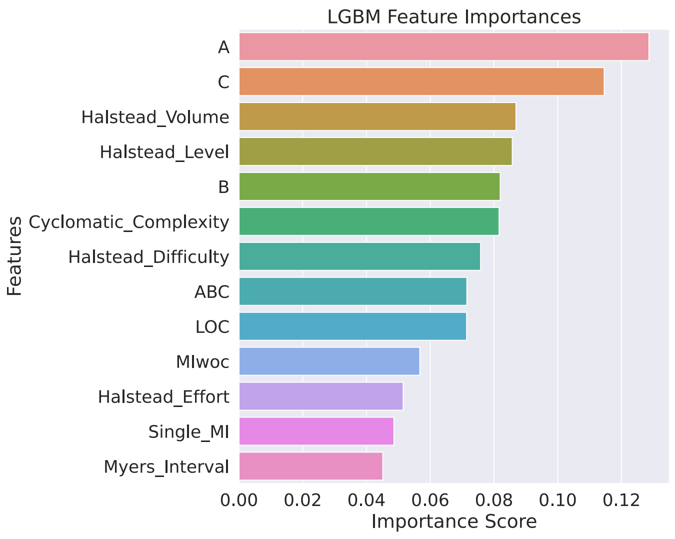


Fig. 5. Feature importances for LGBM, 80/20 train test split.

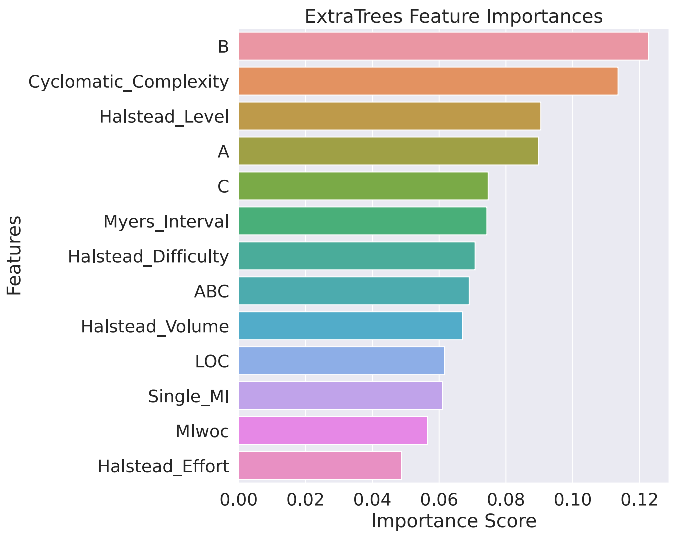


Fig. 6. Feature importances for ExtraTrees, program-wise train test split.

5.2. Explainability

The final part of our analysis aims to analyze, interpret, and explain our results through a feature importance analysis and SHAP values. This approach goes beyond the initial analysis of classification scores and confusion matrices, enabling us to deduce the relationships between features within our classification process and furthering the conversation on the complexities introduced by obfuscation layering.

For the two algorithms that outperformed others—each representing the best for a particular split—we discuss the results within the main text. The feature importance for these models is illustrated in Figs. 5 and 6, with a corresponding selected SHAP values analysis detailed in Figs. 7 and 8. For a more extensive review, including the analyses of additional classifiers, we refer to Appendices B.2 and B.3.

Building on our prior discussion regarding the complexity introduced by various obfuscation techniques, our investigation reveals that different complexity metrics carry varying degrees of importance in our classification process. As shown in Figs. 5 and B.15, features labeled `A`, `B`, and `C` consistently rank among the top five in terms of importance. This observation holds across most classifiers (refer to Appendix B.2) with a few exceptions. Additionally, cyclomatic complexity stands as a top-five feature in five out of the six models reviewed. Halstead-based metrics display more variation in their importance, with `Halstead_Level` appearing three times and `Halstead_Volume` twice within the top five across various analyses. The SHAP values, which we examine next, echo these findings and reveal further interconnections.

Looking at the SHAP values for both of our best classifiers, as displayed in Figs. 7 and 8,[5] we encounter a consistent theme. The plots categorize each feature according to its importance for the model to correctly classify a given class or obfuscation. Specifically, for well-classified categories such as `No-Obfuscation`, `Encode`, `Virtualize`, and `VirtualizeSplit`, we find that the `B` feature is crucial for accurate classification. This finding aligns with the earlier feature importance analysis, detailed in Figs. 3 and 6. In those analyses, `B` emerges as the top feature for the program-wise split and consistently ranks in the top five for importance. This pattern holds across the board for other classifiers and their respective SHAP values, as outlined in Appendix B.3, where `B` invariably appears as a key feature across classes.

[5] Here we show selected classes important for this analysis. The full spectrum is collected in Appendix B.3.

Turning to the implications of obfuscation layering, it is crucial to examine the three clusters where classes are frequently misidentified, as well as the classes with the highest performance, as depicted in Figs. 5 and B.15. In these instances, the `B` feature emerges as the most significant, ranking as the top influencer for the program-wise split and consistently appearing among the top five for other splits. This pattern holds across all classifiers and is reflected in their SHAP values, as detailed in Appendix B.3, where `B` is invariably the most crucial feature for classes such as `No-Obfuscation` and `Encode`, to classes which we can identify with high accuracy. Consequently, we deduce that `B` stands out as the paramount code metric for distinguishing between obfuscated and non-obfuscated code, a conclusion supported by its prominence across various classes in the SHAP value analyses.

Examining our three clusters where obfuscation layers often lead to misclassification, we notice that `B` emerges as the most critical metric in all SHAP value analyses for `FlattenSplit` and `FlattenSplitEncode`. However, the significance of other code metrics varies, with cyclomatic complexity frequently ranking as one of the most influential. In contrast, for `OpaFlatten` and `OpaSplit`, no single feature consistently dominates, indicating that these classes are particularly challenging to distinguish. This suggests that our current set of code complexity metrics might not adequately represent the structural changes of these transformations. In the third cluster, involving `Split` and `SplitFlatten`, where we also see weaker performance and frequent misclassifications, `B` remains a key feature. Interestingly, among the 12 SHAP plots available for these classes, `Halstead_Volume` is identified as the second most crucial metric in seven instances, implying its relevance in classifying these categories accurately, albeit not sufficiently on its own.

Furthermore, we note that `Myers_Interval` and `Halstead_Effort` frequently rank at the lower end of feature importance according to our SHAP value analysis. This trend is consistent with our feature importance findings for both the top classifiers and the additional analyses provided in Appendix B.2. However, it is noteworthy that `Myers_Interval` is often identified as the least significant feature in the program-wise split, yet it assumes greater importance in the 80/20 split. This discrepancy might seem contradictory, especially since our 80/20 splits generally underperform compared to the program-wise splits. From this observation, we deduce that `Myers_Interval` may not effectively capture the intricacies of obfuscations and layering, underscoring its limitations as a metric for distinguishing obfuscated code.

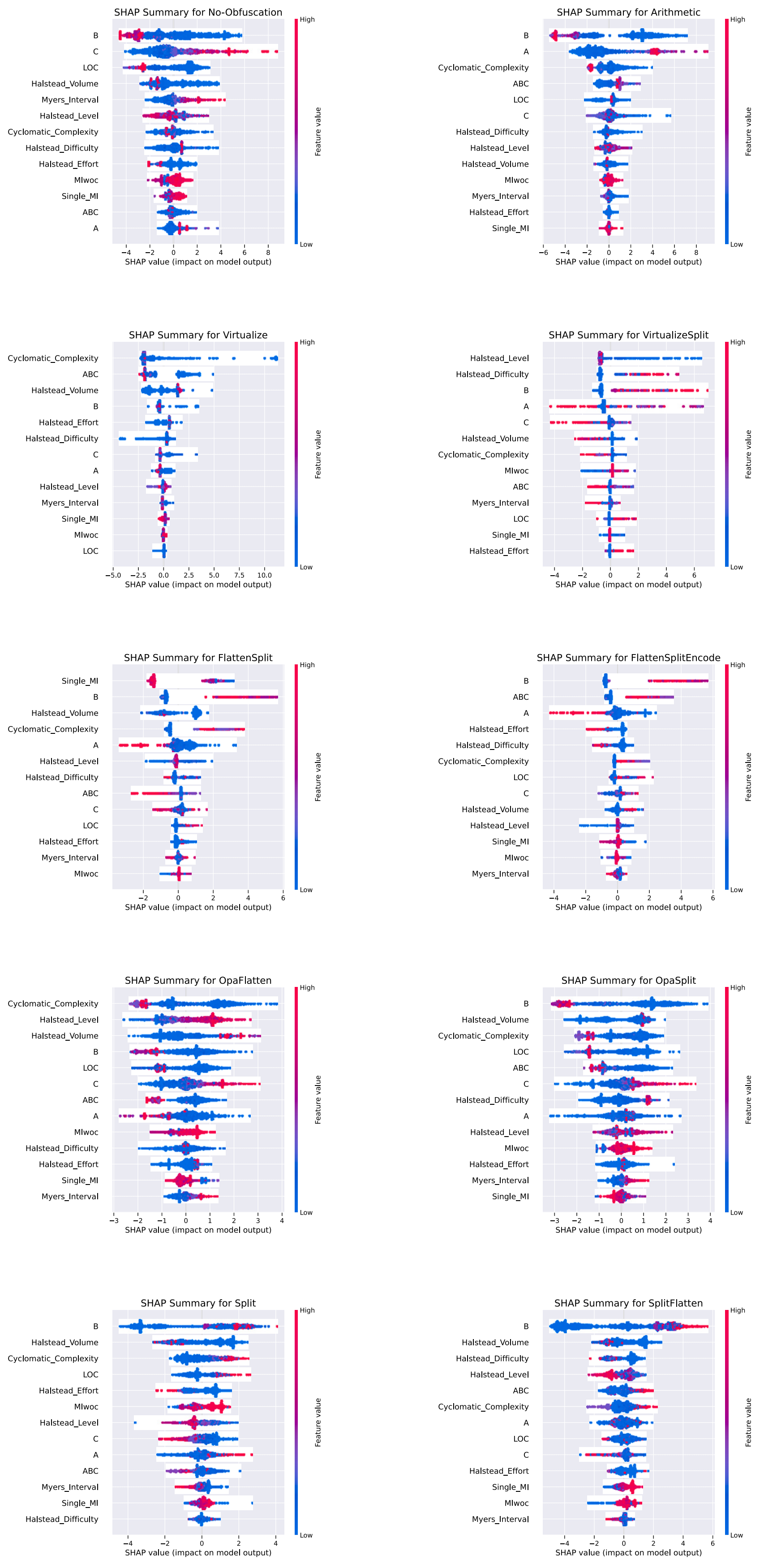


Fig. 7. SHAP value plots, selected classes, LGBM-Classifier, 80/20 split.

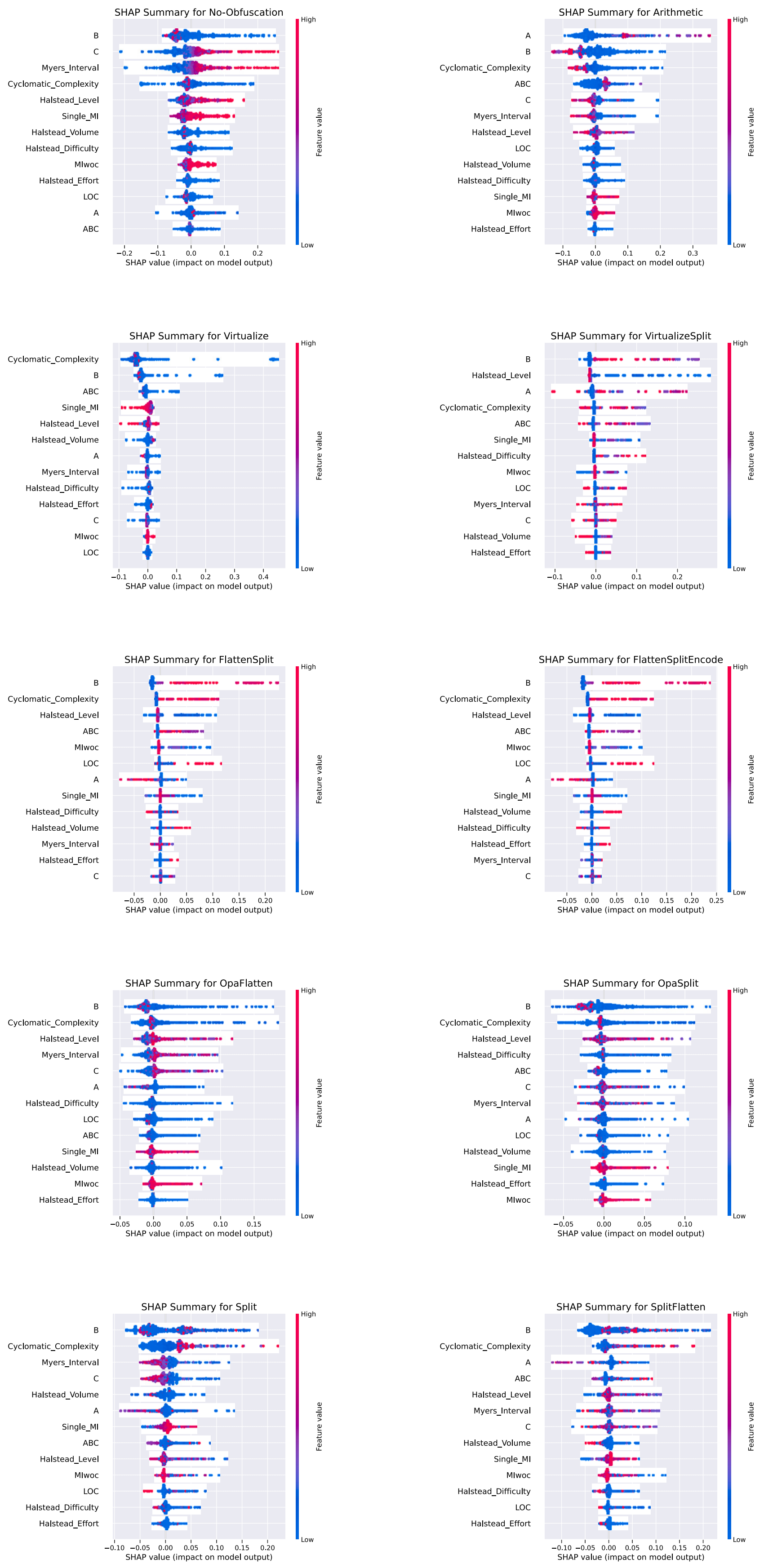


Fig. 8. SHAP value plots, selected classes, extratrees-Classifier, program-wise split.

5.3. Summary

In wrapping up our analysis of Machine Learning experiments focused on code obfuscation, we briefly revisit our methodology and key discoveries with respect to the outlined research objectives of this article. Our exploration spanned various obfuscation techniques, employing three principal algorithms across two distinct data splits—the traditional 80/20 split and a more nuanced program-wise split. This approach allowed us to rigorously evaluate the efficacy of code complexity metrics in identifying specific obfuscations and their layerings. Specifically, we show how code complexity metrics can be used as features to identify obfuscations and their layerings. We identify which complexity metrics are the most expressive for recognizing obfuscated code and examine if and how the layering of obfuscations complicates their identification. ,

Summary of Main Findings:

- **Algorithm Performance Under Different Splits:** The LGBM and ExtraTrees algorithms demonstrate superior performance under the 80/20 and program-wise splits, respectively. This variance in effectiveness emphasizes the significant role of train–test split choice on model training and validation. The program-wise split, by including all versions of obfuscated programs within the same subset, provides a more holistic dataset, potentially enhancing the learning environment and, by extension, model generalization and accuracy.
- **Impact on Model Generalization:** Our findings suggest that data structuring, influenced by the train–test splitting strategy, plays a crucial role in algorithm performance. The differences in cross-validation scores between splits indicate that no algorithm universally excels, pointing to the importance of dataset composition and split strategy in achieving optimal model performance.
- **Obfuscation Identification:** We successfully pinpointed types of obfuscations and their combinations that can be accurately identified. This means that, based on code complexity metrics, we are capable of discerning if and how, through specific techniques and layering, the employed obfuscation can be detected with high precision.
- **Obfuscation Clusters:** We identified three clusters of obfuscation techniques that tend to be confused with one another, leading to clusters characterized by low accuracy. Contrary to our initial assumption, the order of obfuscations in the layering process has only a minor impact on classification accuracy.
- **Feature Significance:** The B feature emerged as particularly pivotal, significantly aiding in differentiating between obfuscated and non-obfuscated code. This reinforces the idea that not all metrics carry equal weight in the identification process.
- **Metric Efficacy:** The analysis underscored that no single code complexity metric is fully capable of capturing the breadth of obfuscation techniques, suggesting a need for more refined or additional metrics to improve detection rates.
- **Data Split Impact:** The importance of `Myers_Interval` varied notably between the two data splits, hinting at its limited reliability as a metric for obfuscation identification. This variability points to the challenges inherent in using static metrics to detect dynamic coding practices.

6. Conclusions

In this work, we provided novel insights into the impact of obfuscation layering on the code structure of protected binaries and the limitations of obfuscation classification due to layering.

Our initial hypothesis suggested that the order of applied code transformations plays a crucial role in the structure of the resulting binary code. While we assumed that later applied obfuscations would obscure earlier ones if they modified similar structural characteristics, our findings indicate that this hypothesis does not hold completely. Instead, the contained transformations, even if applied earlier in the layering process, are crucial for classification and later applied transformations do not undo their characteristic structural patterns enough to prevent classification.

The results of our research can be used to improve the stealth of future protection methodologies. By understanding which obfuscation techniques and combinations are most effective at concealing structural patterns of their individual obfuscation, developers can create obfuscation strategies that are harder to classify through machine-learning.

CRediT authorship contribution statement

Sebastian Raubitzek: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Methodology, Investigation, Formal analysis, Data curation. **Sebastian Schrittwieser:** Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Elisabeth Wimmer:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation. **Kevin Mallinger:** Writing – review & editing, Validation, Supervision, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We shared a link to the data in the paper.

Acknowledgments

The financial support by the Austrian Federal Ministry of Labour and Economy, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

This research was funded in part, by the Austrian Science Fund (FWF) I 3646-N31. For the purpose of open access, the author has applied a CC BY public copyright license to any Author Accepted Manuscript version arising from this submission.

Appendix A. Tigress configurations

```
- CFG flattening (Flatten):
    --Transform=Flatten \
        --Functions=init_program

- Opaque predicates, anti branch analysis, MBA (Opabaea):
    --Seed=0 \
    --Inputs='+1:int:42,-1:length:1?10' \
    --Transform=InitEntropy \
        --Functions=init_program \
        --InitEntropyKinds=vars \
    --Transform=InitOpaque \
        --Functions=init_program \
        --InitOpaqueStructs=list,array,input,env \
    --Transform=InitBranchFuns \
        --InitBranchFunsCount=1 \
    --Transform=AddOpaque \
        --Functions=init_program \
        --AddOpaqueStructs=list \
        --AddOpaqueKinds=true \
    --Transform=AntiBranchAnalysis \
        --Functions=init_program \
        --AntiBranchAnalysisKinds=branchFuns \
        --AntiBranchAnalysisObfuscateBranchFunCall=false \
        --AntiBranchAnalysisBranchFunFlatten=true \
    --Transform=EncodeArithmetic \
        --Functions=init_program

- Virtualization (Virtualize):
    --Transform=Virtualize \
        --VirtualizeDispatch=direct \
        --Functions=init_program

- MBA (Encode):
    --Transform=EncodeArithmetic \
        --Functions=init_program

- Function splitting, MBA (SplitEncode):
    --Transform=Split \
        --SplitKinds=deep,block,top \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=block \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=EncodeArithmetic \
        --Functions=init_program

- Function splitting (Split):
    --Transform=Split \
        --SplitKinds=deep,block,top \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=block \
        --SplitCount=100 \
        --Functions=init_program

- CFG flattening, function splitting (FlattenSplit):
    --Transform=Flatten \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=deep,block,top \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=block \
        --SplitCount=100 \
        --Functions=init_program

- Function splitting, CFG flattening (SplitFlatten):
    --Transform=Split \
        --SplitKinds=deep,block,top \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=block \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Flatten \
        --Functions=init_program

- CFG flattening, function splitting, MBA (FlattenSplitEncode):
    --Transform=Flatten \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=deep,block,top \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=block \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=EncodeArithmetic \
        --Functions=init_program

- Virtualization, function splitting (VirtualizeSplit):
    --Transform=Virtualize \
        --VirtualizeDispatch=direct \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=deep,block,top \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=block \
        --SplitCount=100 \
        --Functions=init_program

- Function splitting, virtualization (SplitVirtualize):
    --Transform=Split \
        --SplitKinds=deep,block,top \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=block \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Virtualize \
        --VirtualizeDispatch=direct \
        --Functions=init_program

- Opaque predicates (Opa):
    --Seed=0 \
    --Inputs='+1:int:42,-1:length:1?10' \
    --Transform=InitEntropy \
        --Functions=init_program \
        --InitEntropyKinds=vars \
    --Transform=InitOpaque \
        --Functions=init_program \
        --InitOpaqueStructs=list,array,input,env \
    --Transform=InitBranchFuns \
        --InitBranchFunsCount=1 \
    --Transform=AddOpaque \
        --Functions=init_program \
        --AddOpaqueStructs=list \
        --AddOpaqueKinds=true

- Function splitting, opaque predicates (SplitOpa):
    --Transform=Split \
        --SplitKinds=deep,block,top \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=block \
        --SplitCount=100 \
        --Functions=init_program \
    --Seed=0 \
    --Inputs='+1:int:42,-1:length:1?10' \
    --Transform=InitEntropy \
        --Functions=init_program \
        --InitEntropyKinds=vars \
    --Transform=InitOpaque \
        --Functions=init_program \
        --InitOpaqueStructs=list,array,input,env \
    --Transform=InitBranchFuns \
        --InitBranchFunsCount=1 \
    --Transform=AddOpaque \
        --Functions=init_program \
        --AddOpaqueStructs=list \
        --AddOpaqueKinds=true

- Opaque predicates, function splitting (OpaSplit):
    --Seed=0 \
    --Inputs='+1:int:42,-1:length:1?10' \
    --Transform=InitEntropy \
        --Functions=init_program \
        --InitEntropyKinds=vars \
    --Transform=InitOpaque \
        --Functions=init_program \
        --InitOpaqueStructs=list,array,input,env \
    --Transform=InitBranchFuns \
        --InitBranchFunsCount=1 \
    --Transform=AddOpaque \
        --Functions=init_program \
        --AddOpaqueStructs=list \
        --AddOpaqueKinds=true \
    --Transform=Split \
        --SplitKinds=deep,block,top \
        --SplitCount=100 \
        --Functions=init_program \
    --Transform=Split \
        --SplitKinds=block \
        --SplitCount=100 \
        --Functions=init_program
```

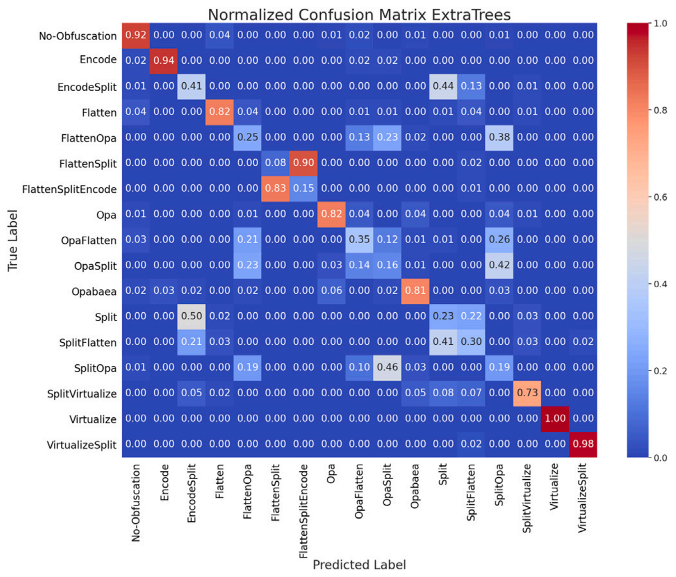


Fig. B.9. Confusion matrix for ExtraTrees, 80/20 train test split.

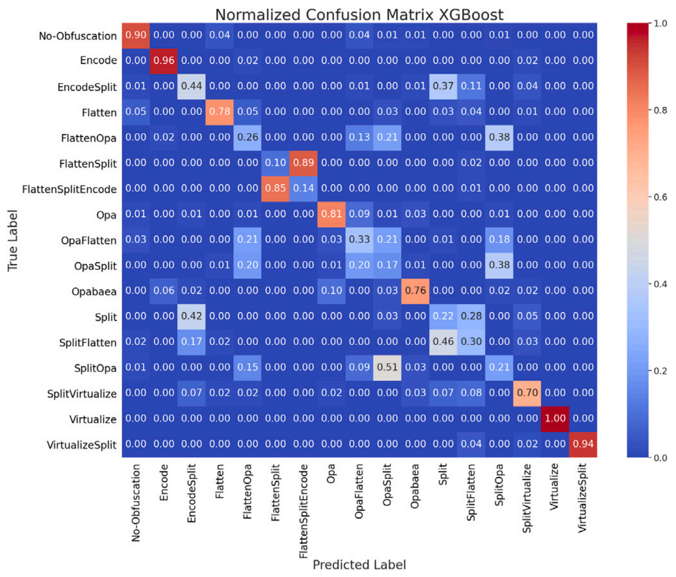


Fig. B.10. Confusion matrix for XGBoost, 80/20 train test split.

```
- Opaque predicates, CFG flattening (OpaFlatten):
    --Transform=Flatten \
        --Functions=init_program \
    --Seed=0 \
    --Inputs='+1:int:42,-1:length:1?10' \
    --Transform=InitEntropy \
        --Functions=init_program \
        --InitEntropyKinds=vars \
    --Transform=InitOpaque \
        --Functions=init_program \
        --InitOpaqueStructs=list,array,input,env \
    --Transform=InitBranchFuns \
        --InitBranchFunsCount=1 \
    --Transform=AddOpaque \
        --Functions=init_program \
        --AddOpaqueStructs=list \
        --AddOpaqueKinds=true

- CFG flattening, opaque predicates (FlattenOpa):
    --Transform=Flatten \
        --Functions=init_program \
    --Seed=0 \
    --Inputs='+1:int:42,-1:length:1?10' \
    --Transform=InitEntropy \
        --Functions=init_program \
        --InitEntropyKinds=vars \
    --Transform=InitOpaque \
        --Functions=init_program \
        --InitOpaqueStructs=list,array,input,env \
    --Transform=InitBranchFuns \
        --InitBranchFunsCount=1 \
    --Transform=AddOpaque \
        --Functions=init_program \
        --AddOpaqueStructs=list \
        --AddOpaqueKinds=true
```

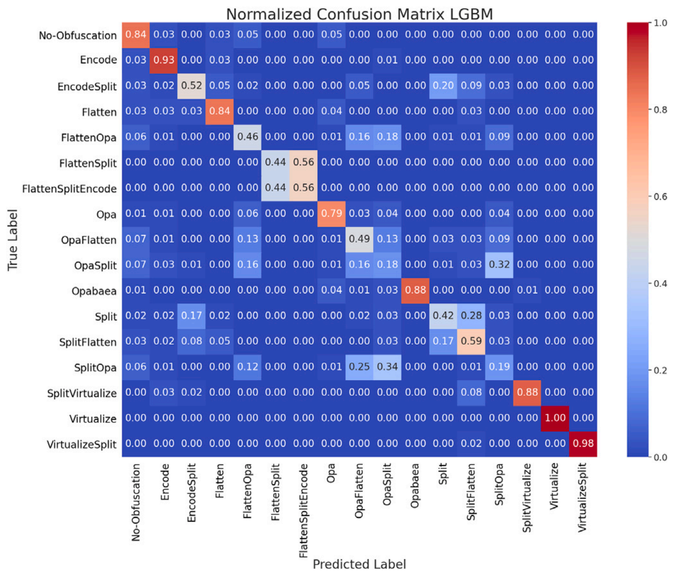


Fig. B.11. Confusion matrix for LGBM, 68/17 program-wise split.

Appendix B. Additional results

This appendix presents the results for all additional setups, i.e., we tested and discussed only the best-performing results in Section 5.1; this appendix serves to present the remaining results. Thus, we provide the full spectrum of our analysis.

B.1. Confusion matrices

See Figs. B.9–B.12.

B.2. Feature importances

See Figs. B.13–B.16.

B.3. SHAP value plots

See Figs. B.17–B.22.

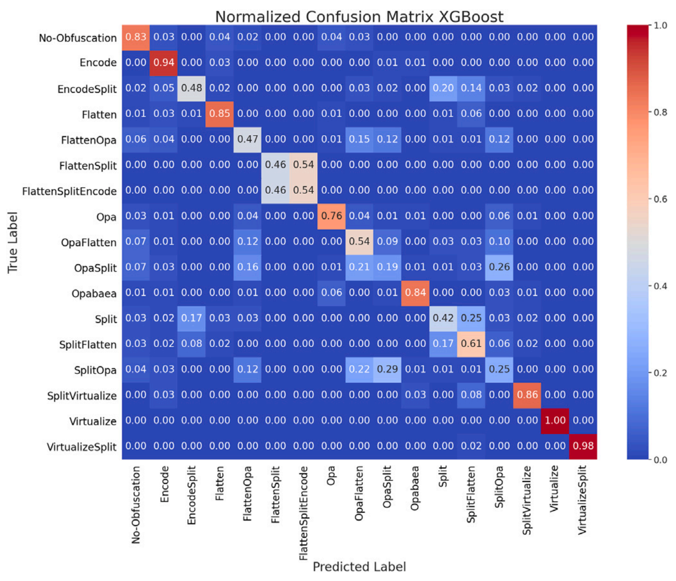


Fig. B.12. Confusion matrix for XGBoost, 68/17 program-wise split.

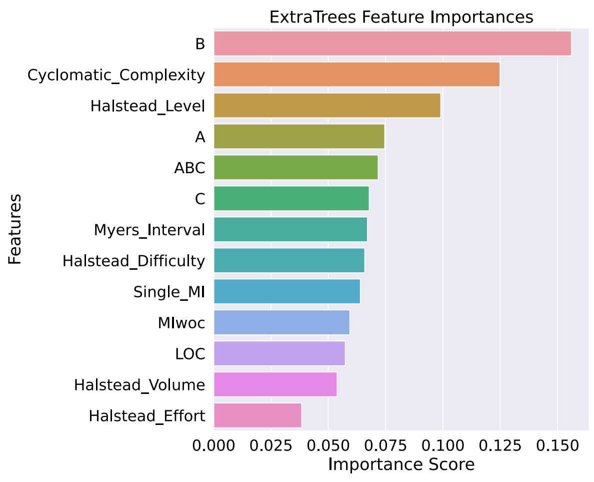


Fig. B.13. Feature importances for ExtraTrees, 80/20 train test split.

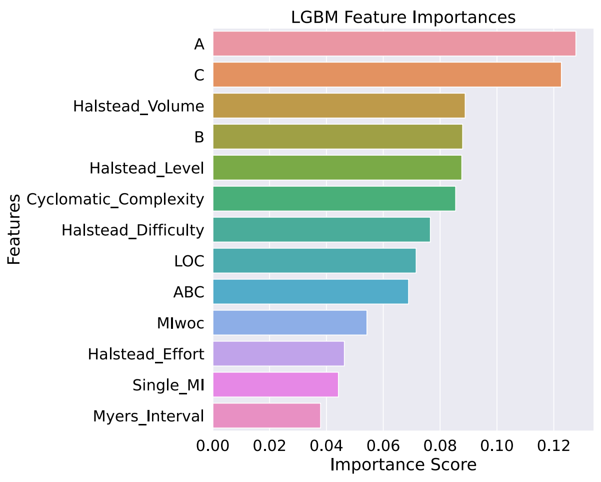


Fig. B.15. Feature importances for LGBM, 68/17 program-wise split.

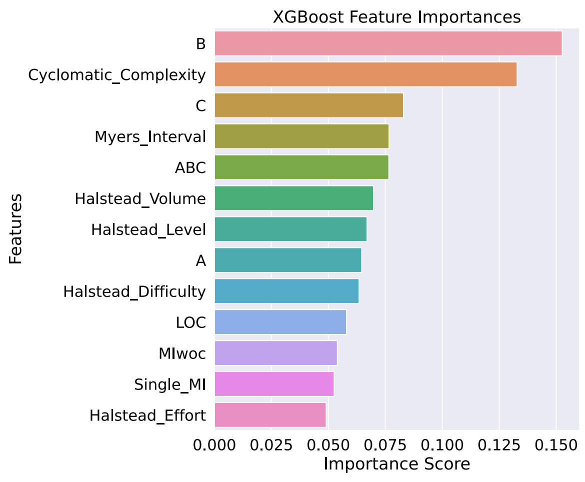


Fig. B.14. Feature importances for XGBoost, 80/20 train test split.

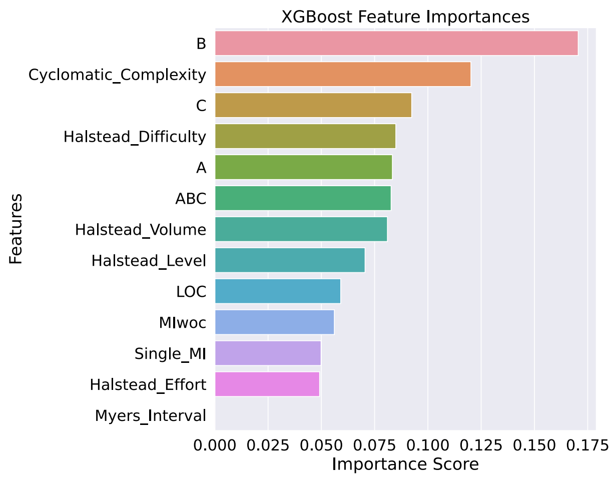


Fig. B.16. Feature importances for XGBoost, 68/17 program-wise split.

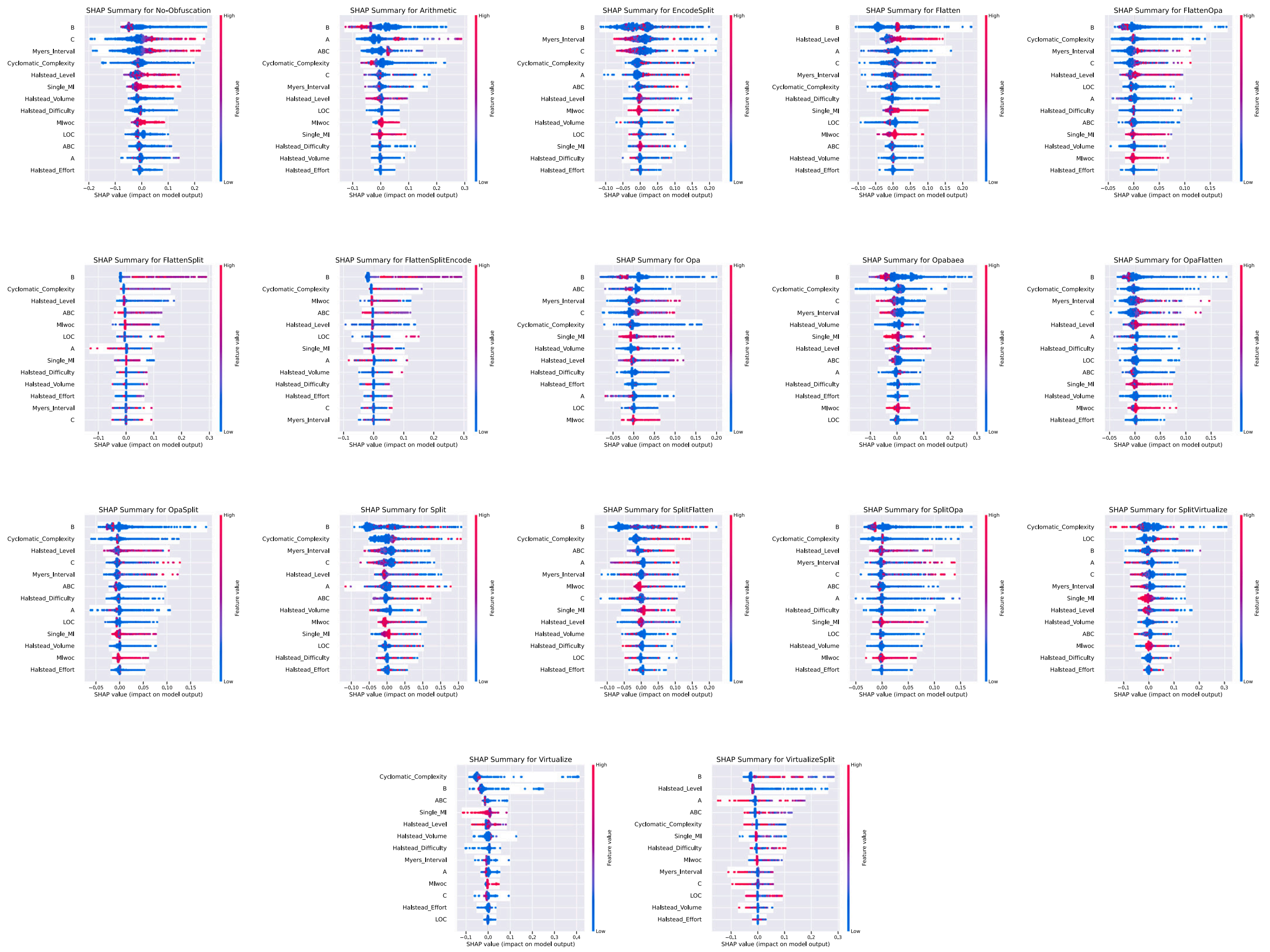


Fig. B.17. SHAP value plots, all classes, ExtraTrees-Classifier, 80/20 split.

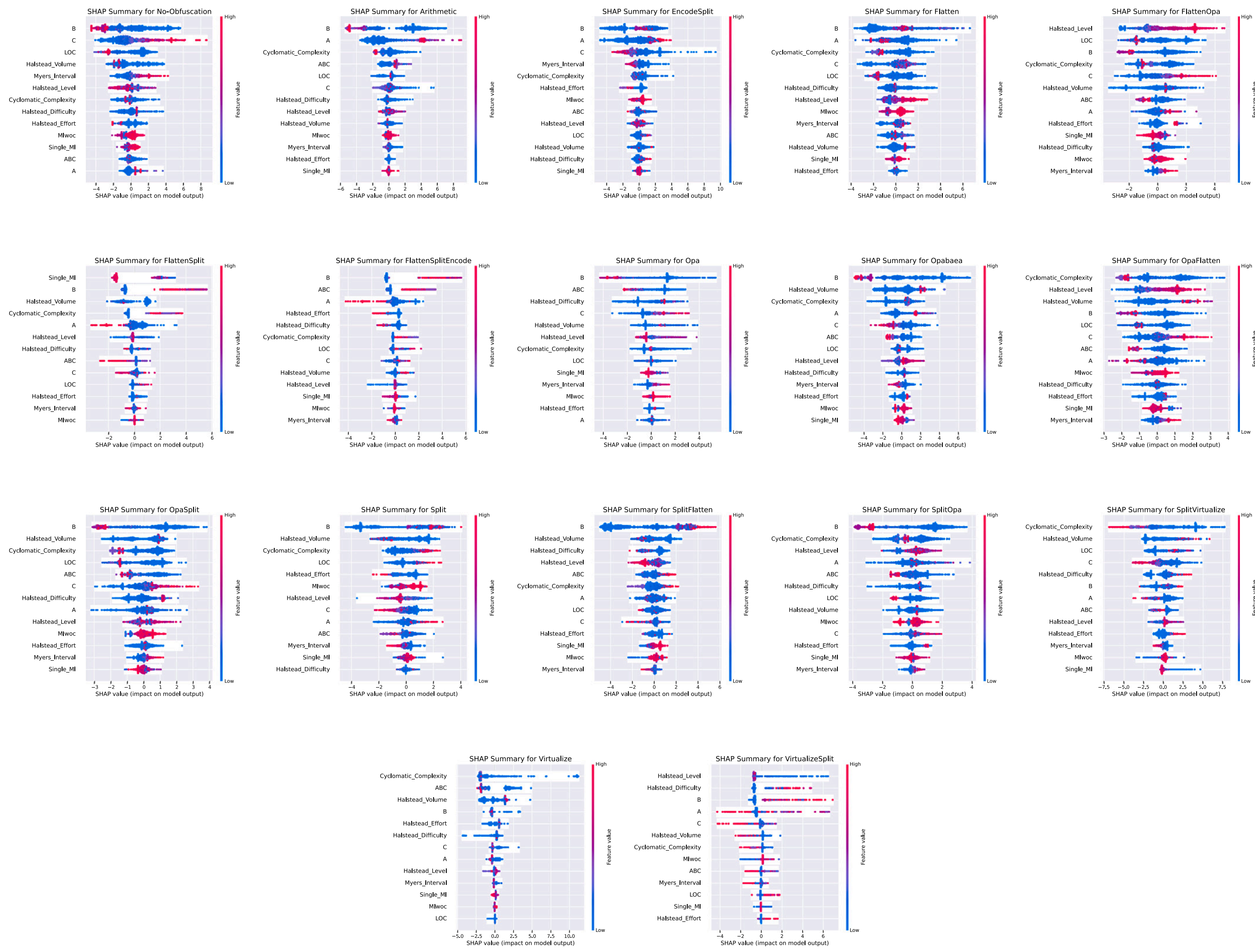


Fig. B.18. SHAP value plots, all classes, LGBM-Classifier, 80/20 split.

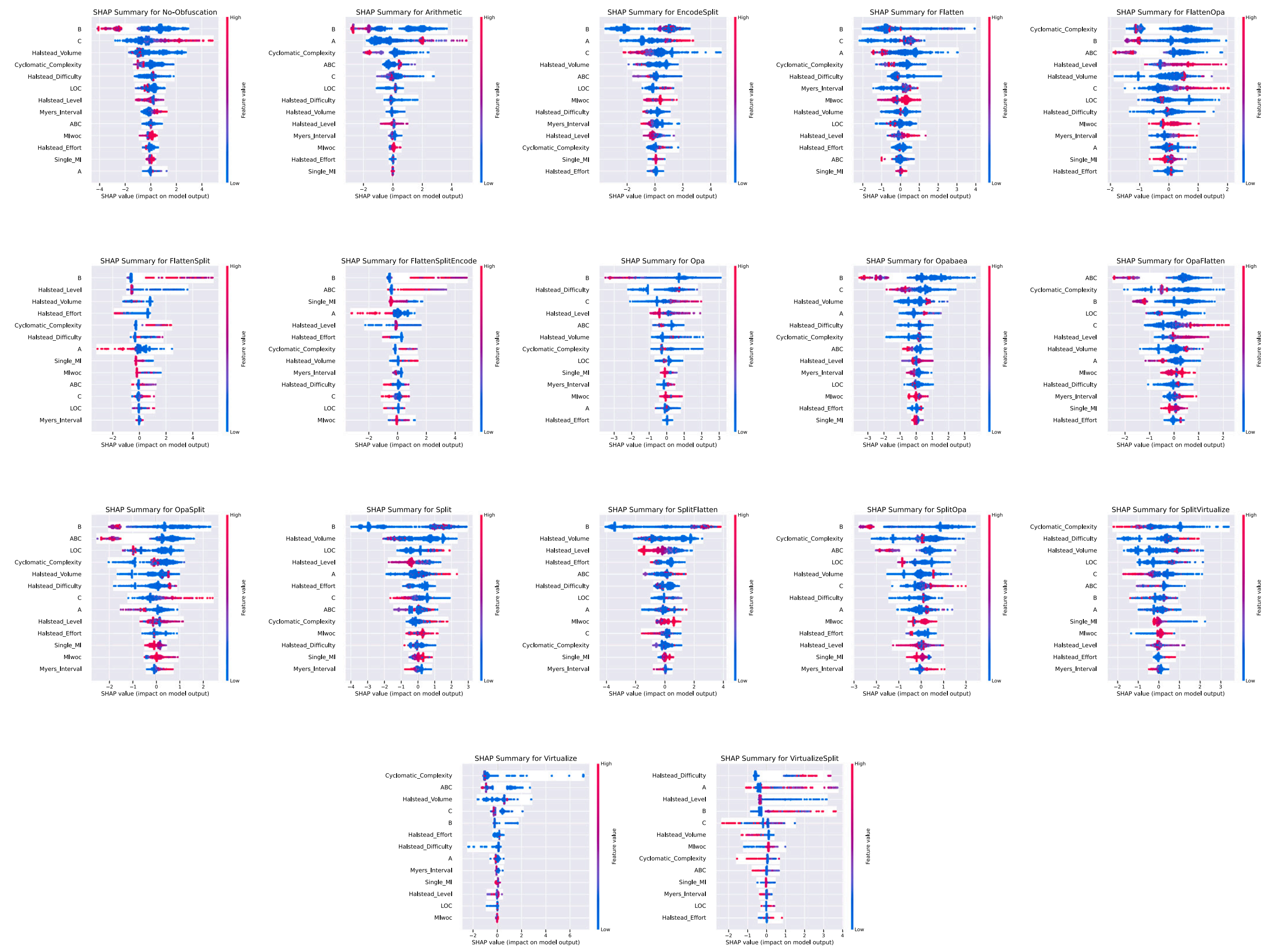


Fig. B.19. SHAP value plots, all classes, XGBoost-Classifier, 80/20 split.

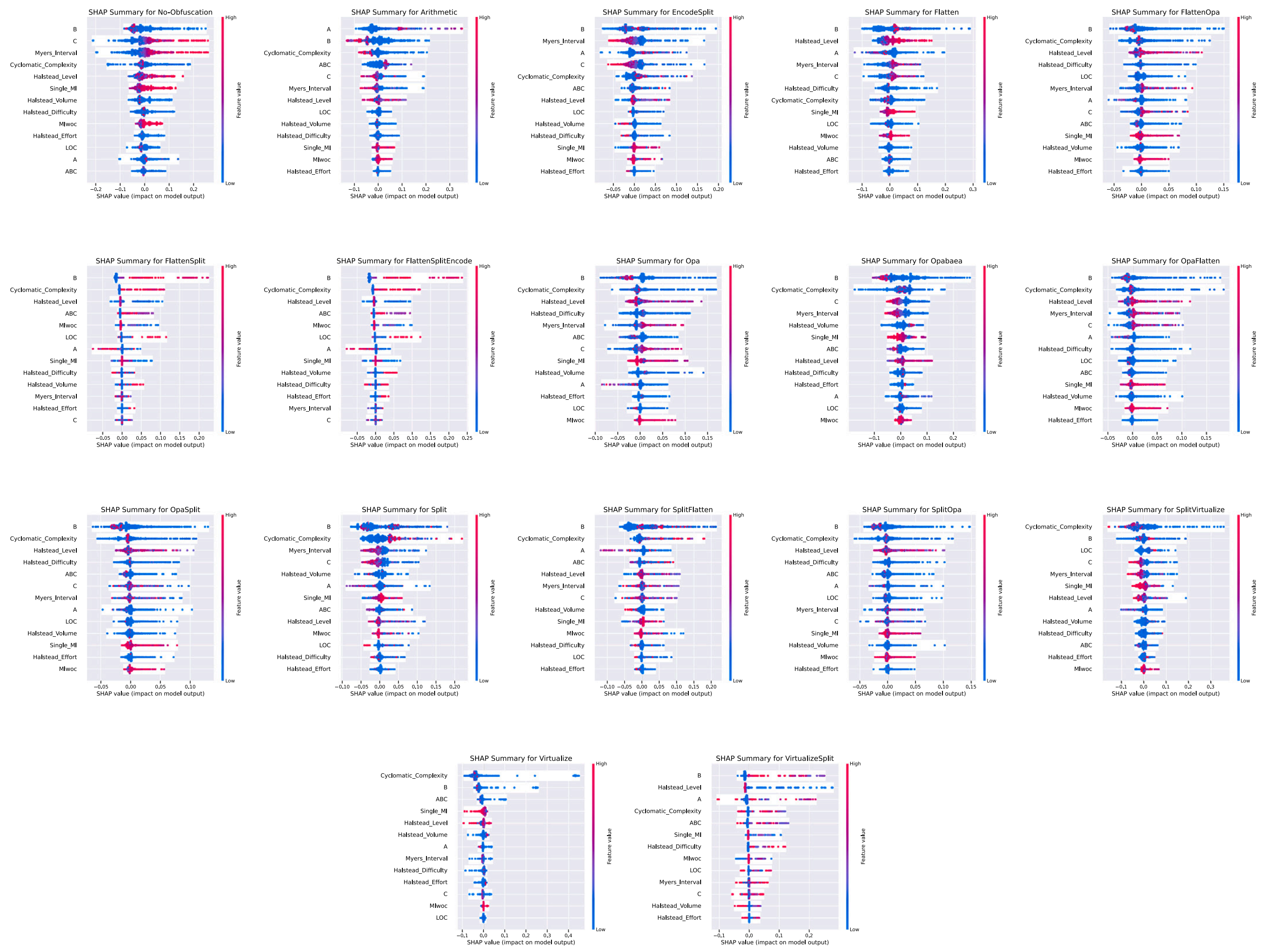


Fig. B.20. SHAP value plots, all classes, ExtraTrees-Classifier, program-wise split.

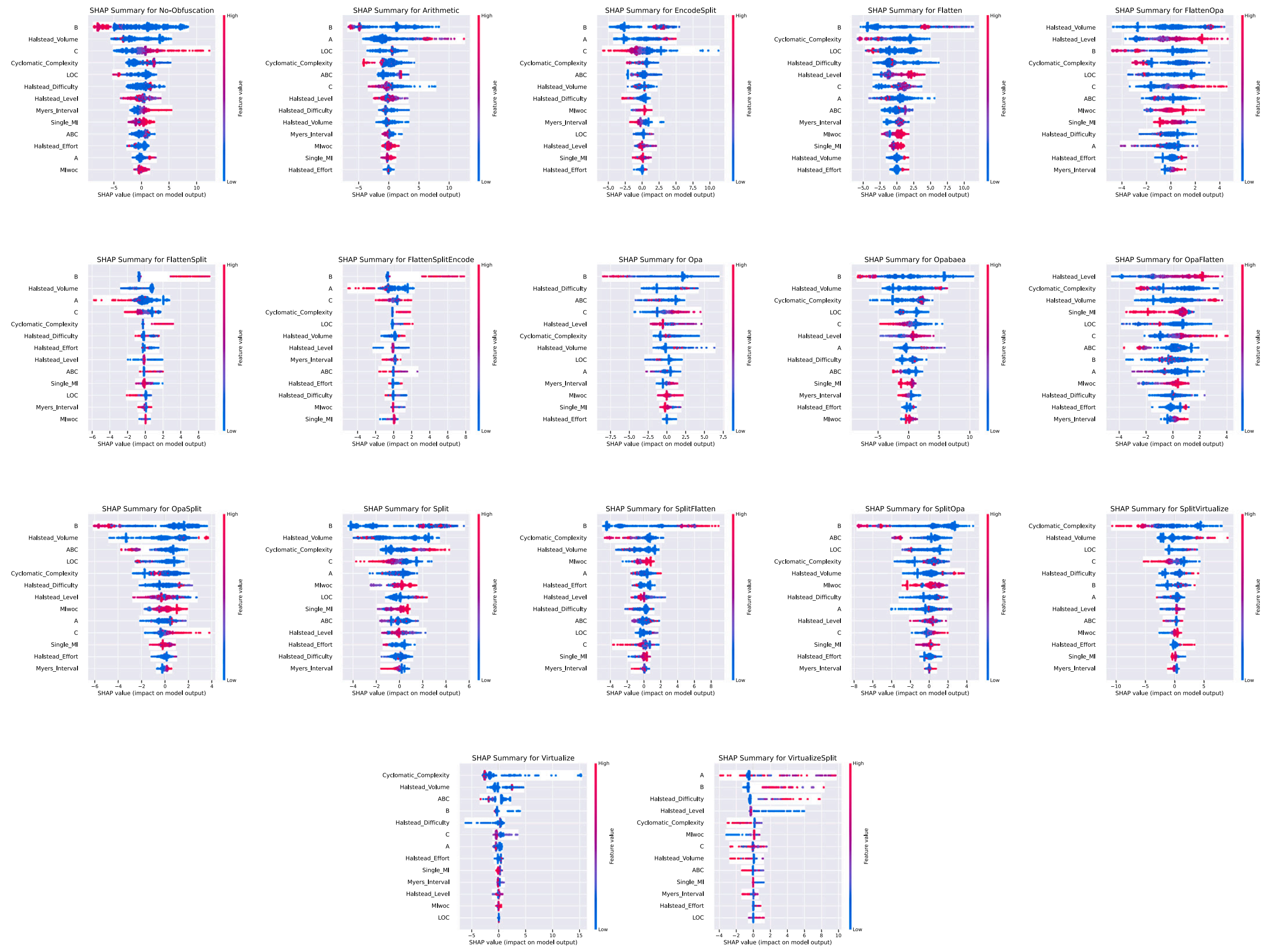


Fig. B.21. SHAP value plots, all classes, LGBM-Classifier, program-wise split.

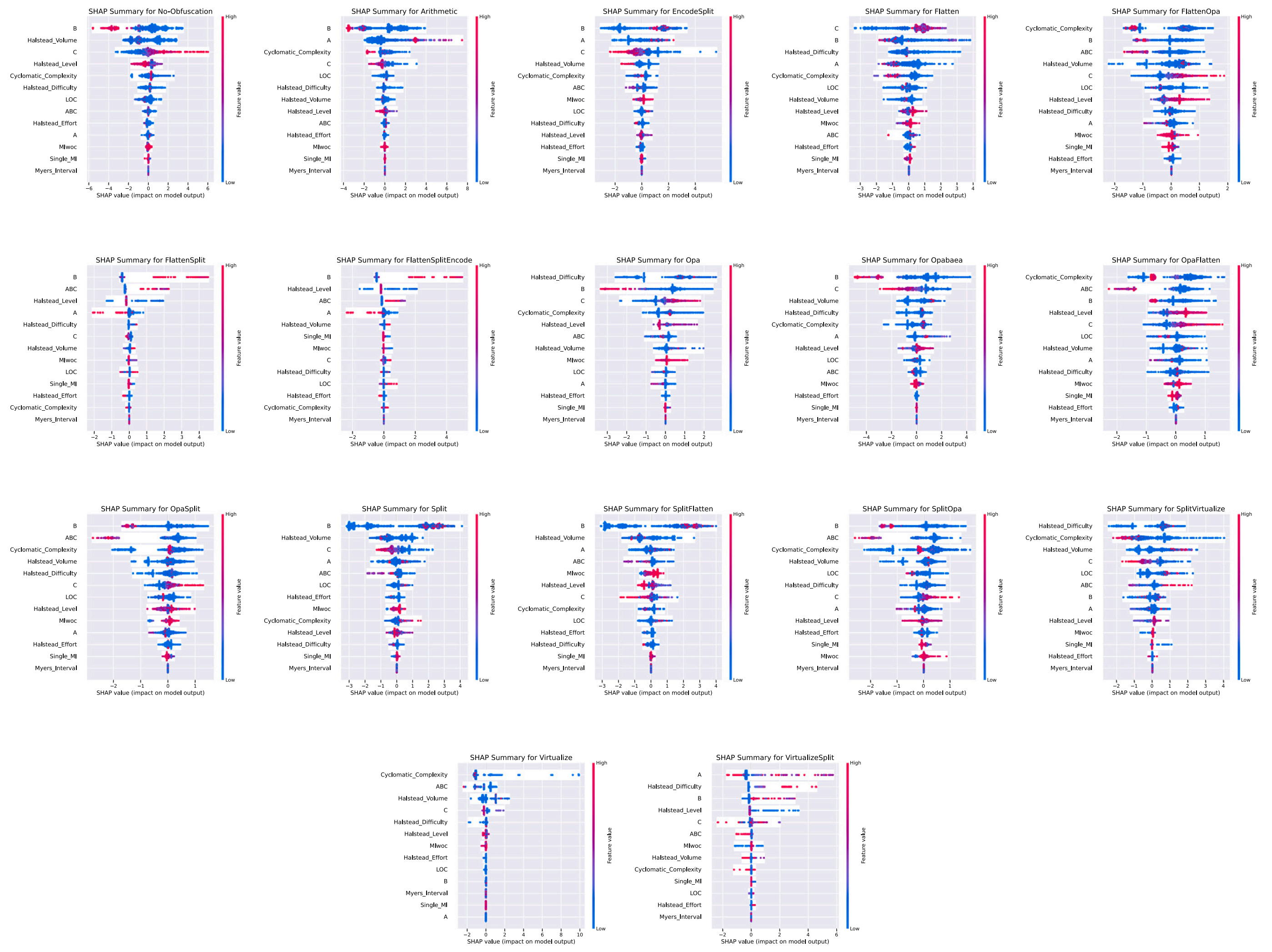


Fig. B.22. SHAP value plots, all classes, XGBoost-Classifier, program-wise split.

References

[1] Brosch T, Morgenstern M. Runtime packers: The hidden problem. Black Hat USA; 2006.

[2] 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. IEEE; 2017, p. 391–402.

[3] Morgenstern M, Pilz H. Useful and useless statistics about viruses and anti-virus programs. In: Proceedings of the CARO workshop. Magdeburg, Germany: AV-Test GmbH; 2010, p. 1–21, Presented at CARO 2010 Helsinki. URL https://www.av-test.org/fileadmin/pdf/publications/caro_2010_avtest_presentation_useful_and_useless_statistics_about_viruses_and_anti-virus_programs.pdf. [Accessed 16 March 2024].

[4] Schrittwieser S, Wimmer E, Mallinger K, Kochberger P, Lawitschka C, Raubitzek S, et al. Modeling obfuscation stealth through code complexity. In: European symposium on research in computer security. Springer; 2023, p. 392–408.

[5] Collberg C, Martin S, Myers J, Nagra J. Distributed application tamper detection via continuous software updates. In: Proceedings of the 28th annual computer security applications conference. 2012, p. 319–28.

[6] 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. 1998, p. 184–96.

[7] Nagra J, Collberg C. Surreptitious software: Obfuscation, watermarking, and tamperproofing for software protection: Obfuscation, watermarking, and tamperproofing for software protection. Pearson Education; 2009.

[8] 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. 2010, p. 536–46.

[9] Mason J, Small S, Monrose F, MacManus G. English shellcode. In: Proceedings of the 16th ACM conference on computer and communications security. 2009, p. 524–33.

[10] 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. IEEE; 2015, p. 31–7.

[11] Wang Y, Rountev A. Who changed you? Obfuscator identification for Android. In: 2017 IEEE/ACM 4th international conference on mobile software engineering and systems. IEEE; 2017, p. 154–64.

[12] 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. 2018, p. 1–9.

[13] Park M, You G, Cho Sj, Park M, Han S. A framework for identifying obfuscation techniques applied to android apps using machine learning.. J Wirel Mob Netw Ubiquitous Comput Dependable Appl 2019;10(4):22–30.

[14] Jones L, Christman D, Banescu S, Carlisle M. Bytewise: A case study in neural network obfuscation identification. In: 2018 IEEE 8th annual computing and communication workshop and conference. IEEE; 2018, p. 155–64.

[15] Kim J, Kang S, Cho ES, Paik JY. 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. Springer; 2021, p. 3–15.

[16] Salem A, Banescu S. Metadata recovery from obfuscated programs using machine learning. In: Proceedings of the 6th workshop on software security, protection, and reverse engineering. 2016, p. 1–11.

[17] Sagisaka H, Tamada H. Identifying the applied obfuscation method towards de-obfuscation. In: 2016 IEEE/ACIS 15th international conference on computer and information science. IEEE; 2016, p. 1–6.

[18] Tesauro GJ, Kephart JO, Sorkin GB. Neural networks for computer virus recognition. IEEE Expert 1996;11(4):5–6.

[19] Sebastian SA, 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. IEEE; 2016, p. 1–6.

[20] Necula GC, McPeak S, Rahul SP, Weimer W. CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool RN, editor. Compiler construction. Berlin, Heidelberg: Springer Berlin Heidelberg; 2002, p. 213–28.
[21] 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. CSREA Press; 2006, p. 527–33.
[22] 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. IEEE; 2015, p. 3–9.
[23] Kim H, Park J, Kwon H, Jang K, Seo H. Convolutional neural network-based cryptography ransomware detection for low-end embedded processors. Mathematics 2021;9(7):705.
[24] Halstead MH. Elements of software science. Operating and programming systems series, USA: Elsevier Science Inc.; 1977.
[25] McCabe TJ. A complexity measure. IEEE Trans Softw Eng 1976;(4):308–20.
[26] Ikerionwu C. Cyclomatic complexity as a software metric. Int J Acad Res 2010;2(3).
[27] Sellers BH. Modularization and McCabe's cyclomatic complexity. Commun ACM 1992;35(12):17–20.
[28] Ebert C, Cain J, Antoniol G, Counsell S, Laplante P. Cyclomatic complexity. IEEE Softw 2016;33(6):27–9.
[29] 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. 2004, p. 391–405.
[30] Sarwar MMS, Shahzad S, Ahmad I. Cyclomatic complexity: The nesting problem. In: Eighth international conference on digital information management. IEEE; 2013, p. 274–9.
[31] Madi A, Zein OK, Kadry S. On the improvement of cyclomatic complexity metric. Int J Softw Eng Appl 2013;7(2):67–82.
[32] 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. Springer; 2017, p. 193–202.
[33] Myers GJ. An extension to the cyclomatic measure of program complexity. SIGPLAN Not 1977;12(10):61–4.
[34] Fitzpatrick J. Applying the ABC metric to C, C++, and Java. Technical report, C++ Report; 1997.
[35] Oman P, Hagemeister J. Metrics for assessing a software system's maintainability. In: Proceedings conference on software maintenance 1992. 1992, p. 337–44.
[36] Oman P, Hagemeister J. Construction and testing of polynomials predicting software maintainability. J Syst Softw 1994;24(3):251–66, Oregon Workshop on Software Metrics.
[37] Coleman D, Oman P, Ash D, Lowther B. Using metrics to evaluate software system maintainability. Computer 1994;27(08):44–9.
[38] Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations. Technical report, New Zealand: Department of Computer Science, The University of Auckland; 1997.
[39] Ebad SA, Darem AA, Abawajy JH. Measuring software obfuscation quality–a systematic literature review. IEEE Access 2021;9:99024–38.
[40] Banescu S, Ochoa M, Pretschner A. A framework for measuring software obfuscation resilience against automated attacks. In: 2015 IEEE/ACM 1st international workshop on software protection. 2015, p. 45–51. http://dx.doi.org/10.1109/SPRO.2015.16.
[41] He H, Bai Y, Garcia EA, Li S. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In: 2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence). 2008, p. 1322–8. http://dx.doi.org/10.1109/IJCNN.2008.4633969.
[42] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine learning in Python. J Mach Learn Res 2011;12:2825–30.
[43] Lundberg SM, Lee SI. A unified approach to interpreting model predictions. In: Proceedings of the 31st international conference on neural information processing systems. Red Hook, NY, USA: Curran Associates Inc.; 2017, p. 4768–77.
[44] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine learning in Python – Feature importances with a forest of trees. 2011, Visited on 16.03.2024. https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html.
[45] Geurts P, Ernst D, Wehenkel L. Extremely randomized trees. Mach Learn 2006;63(1):3–42. http://dx.doi.org/10.1007/s10994-006-6226-1.
[46] Chen T, Guestrin C. XGBoost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. ACM; 2016, p. 785–94. http://dx.doi.org/10.1145/2939672.2939785.
[47] Ke G, Meng Q, Finley T, Wang T, Chen W, Ma W, et al. LightGBM: A highly efficient gradient boosting decision tree. In: Advances in neural information processing systems. 2017, p. 3146–54, URL: https://papers.nips.cc/paper/2017/hash/f491e7e8f3e446359e5ed5047b8cfb0c-Abstract.html.
[48] Peng Y, Zhao S, Zeng Z, Hu X, Yin Z. LGBMDF: A cascade forest framework with LightGBM for predicting drug-target interactions. Front Microbiol 2023;13. http://dx.doi.org/10.3389/fmicb.2022.1092467, URL: https://www.frontiersin.org/journals/microbiology/articles/10.3389/fmicb.2022.1092467.
[49] Saeed U, Jan SU, Lee YD, Koo I. Fault diagnosis based on extremely randomized trees in wireless sensor networks. Reliab Eng Syst Saf 2021;205:107284. http://dx.doi.org/10.1016/j.ress.2020.107284, URL: https://www.sciencedirect.com/science/article/pii/S095183202030781X.
[50] Raubitzek S, Mallinger K. On the applicability of quantum machine learning. Entropy 2023;25(7). http://dx.doi.org/10.3390/e25070992, URL: https://www.mdpi.com/1099-4300/25/7/992.
[51] Shehadeh A, Alshboul O, Al Mamlook RE, Hamedat O. Machine learning models for predicting the residual value of heavy construction equipment: An evaluation of modified decision tree, LightGBM, and XGBoost regression. Autom Constr 2021;129:103827. http://dx.doi.org/10.1016/j.autcon.2021.103827, URL: https://www.sciencedirect.com/science/article/pii/S0926580521002788.
[52] Pandala S. Lazy predict. 2021, https://github.com/shankarpandala/lazypredict.
[53] Snoek J, Larochelle H, Adams RP. Practical bayesian optimization of machine learning algorithms. Adv Neural Inf Process Syst 2012;25.