machine learning & knowledge extraction

MDPI

Article

Classification of Obfuscation Techniques in LLVM IR: Machine Learning on Vector Representations

Sebastian Raubitzek [1], Patrick Felbauer [2], Kevin Mallinger [1] and Sebastian Schrittwieser [2,*]

[1] SBA Research gGmbH, Floragasse 7/5.OG, 1040 Vienna, Austria; sraubitzek2@sba-research.org (S.R.); kmallinger@sba-research.org (K.M.)
[2] Christian Doppler Laboratory for Assurance and Transparency in Software Protection, Faculty of Computer Science, University of Vienna, Kolingasse 14–16, 1090 Vienna, Austria; patrick.felbauer@univie.ac.at
* Correspondence: sebastian.schrittwieser@univie.ac.at

Abstract

We present a novel methodology for classifying code obfuscation techniques in LLVM IR program embeddings. We apply isolated and layered code obfuscations to C source code using the Tigress obfuscator, compile them to LLVM IR, and convert each IR code representation into a numerical embedding (vector representation) that captures intrinsic characteristics of the applied obfuscations. We then use two modern boost classifiers to identify which obfuscation, or layering of obfuscations, was used on the source code from the vector representation. To better analyze classifier behavior and error propagation, we employ a staged, cascading experimental design that separates the task into multiple decision levels, including obfuscation detection, single-versus-layered discrimination, and detailed technique classification. This structured evaluation allows a fine-grained view of classification uncertainty and model robustness across the inference stages. We achieve an overall accuracy of more than 90% in identifying the types of obfuscations. Our experiments show high classification accuracy for most obfuscations, including layered obfuscations, and even perfect scores for certain transformations, indicating that a vector representation of IR code preserves distinguishing features of the protections. In this article, we detail the workflow for applying obfuscations, generating embeddings, and training the model, and we discuss challenges such as obfuscation patterns covered by other obfuscations in layered protection scenarios.

Keywords: code-obfuscation; machine learning; obfuscation identification; malware detection; IR2VEC; intermediate representation

Academic Editor: Francesco Buccafurri

Received: 25 August 2025
Revised: 9 October 2025
Accepted: 14 October 2025
Published: 22 October 2025

Citation: Raubitzek, S.; Felbauer, P.; Mallinger, K.; Schrittwieser, S. Classification of Obfuscation Techniques in LLVM IR: Machine Learning on Vector Representations. *Mach. Learn. Knowl. Extr.* **2025**, *7*, 125. https://doi.org/10.3390/make7040125

1. Introduction

Code obfuscation has long been used both by software developers and malware authors alike to make reverse engineering of program code more difficult. Obfuscation methods can be applied at various stages of code representation. Traditionally, they target source code [1], but some protections are also applied directly to binary code. For instance, virtualization obfuscation (e.g., by using tools like VMProtect) typically operates on binaries. Malware packers provide another prime example of binary-level obfuscation. More recently, obfuscation at the intermediate representation (IR) level has also been proposed [2], although it is rarely used in practice.

Regardless of the code representation to which obfuscations are applied, they typically leave characteristic traces in the structure of a program. These characteristics can be

observed both symbolically (e.g., in the distribution of opcodes within binary code) and in flow-based features (e.g., in patterns of control flow graphs). When reverse engineering obfuscated programs, identifying the specific obfuscation techniques applied to code is crucial, as most obfuscations complicate the understanding of program functionality but are not irreversible. With targeted de-obfuscation methods, these transformations can often be at least partially undone.

Previous research on obfuscation identification has already explored various code representations (e.g., source and binary code) and used different code characteristics (code complexity, opcode distribution, etc.) for machine learning-based classification [3–8]. In this article, we focus specifically on identifying obfuscation techniques in the Low Level Virtual Machine intermediate representation (LLVM-IR). We use a dataset of C programs and apply multiple Tigress obfuscations (including layering of multiple obfuscations). Each resulting obfuscated program is then converted to an IR2Vec [9] embedding, which serves as the input for our machine learning pipeline, where CatBoost and ExtraTrees classifiers determine whether and how the code has been obfuscated. This whole workflow is depicted in Figure 1.

To analyze the classification problem in greater detail, we designed a staged and cascading experimental setup. Each stage isolates a specific decision level: first detecting obfuscation, then distinguishing between single and layered transformations, and finally identifying the applied method or combination. This structure enables a more interpretable analysis of model behavior and helps locate where prediction uncertainty emerges across the decision pipeline.

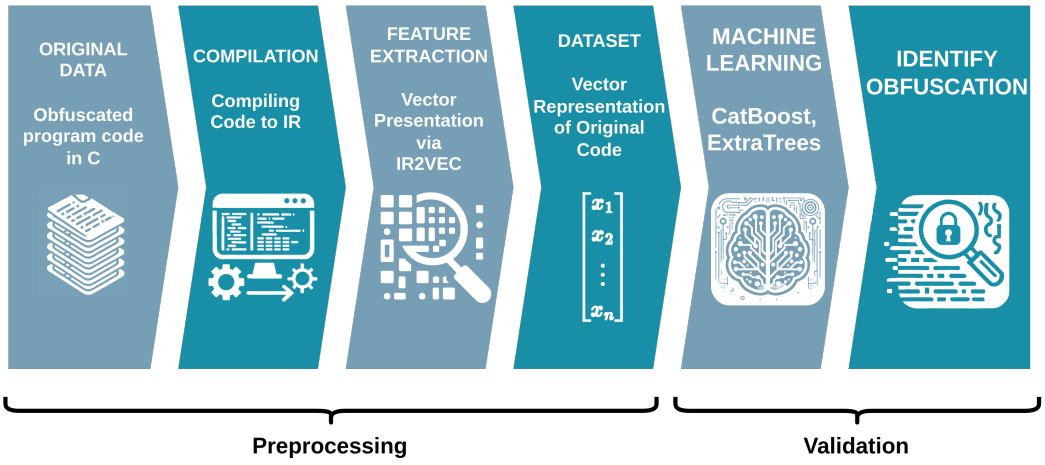

Figure 1. Overview of the obfuscation classification pipeline, from source code transformation with Tigress, conversion of obfuscated C files to LLVM IR, to IR2Vec embedding and subsequent classification using CatBoost and ExtraTrees.

This article is structured as follows: The next section, Section 2, presents work related to our research. Then we discuss all employed obfuscation techniques in Section 3 and present the results of our machine learning experiments in Section 4. We summarize and interpret the findings in Section 5 and present conclusions and directions for future work in Section 6.

2. Related Work

Several studies have employed machine learning on code structure metrics to differentiate between obfuscated and unobfuscated code. For example, Salem et al. [3] applied TF-IDF features to differentiate Tigress-protected C samples. Recent approaches have employed code complexity metrics and grayscale image representations of binary code to

detect Tigress obfuscations [10,11]. A similar method was reported by Raitsis et al. [12], in which entropy-based feature extraction and machine learning were used to determine whether and how a program was obfuscated. Additionally, Jiang et al. [13] demonstrated the effectiveness of convolutional neural networks for identifying obfuscation in code. Sagisaka et al. [4] used birthmark analysis to identify obfuscation tools for Java bytecode.

Research on mobile platforms has also contributed to the field. Wang et al. [14] developed a machine learning method capable of identifying the obfuscation tool, its applied obfuscation type, and configuration in Android applications by extracting feature vectors from Dalvik bytecode, while Bacci et al. [15] derived features from Smali code for similar purposes. Park et al. [16] proposed a framework for detecting class-level obfuscations in Android applications, complementing binary-level approaches such as that introduced by Jones et al.

Our work builds upon these ideas while focusing on IR-level data and analyzing to what extent we can correctly detect obfuscations and layerings thereof.

3. Methodology

This section describes the basic methodology of our classification approach, i.e., obfuscation techniques, the data set, and our machine learning approach.

3.1. Code Obfuscations

In this study, we focus specifically on source-to-source obfuscations using the Tigress obfuscator [17]. In the following, we introduce the obfuscations used in our study.

Control-Flow Transformations

Control-flow transformations alter the possible execution paths of a program to complicate reverse engineering. The *flattening* technique converts the original control flow into a switch-case dispatch mechanism (or any other equivalent control flow dispatching structure), thereby breaking the natural execution sequence and static appearance of the program's control flow graph. *Virtualization* replaces selected functions with code written in a random instruction set that requires a dedicated interpreter bundled with the protected binary. *Bogus control flow* injects false branches (with opaque conditions) that do not affect the runtime behavior of the program but mislead static analysis tools. *Encoding branches* hides the target of branches. *Function splitting* breaks a function into several smaller subfunctions, modifying the program's call graph in a way similar as control-flow obfuscations inside one function.

Data and Arithmetic Transformations

Data and arithmetic transformations modify the representation of variables and constants, as well as arithmetic operations, to complicate program interpretation. *Encoding literals* replaces literal data such as Strings with encoded representations and *arithmetic encoding* transforms arithmetic expressions into more complex ones.

Anti-Analysis Protections

Anti-analysis protections consist of techniques designed specifically to defeat code analysis tools. *Opaque predicates* incorporate conditions that always evaluate to the same result while appearing dynamic to a static analyst. *Anti-taint analysis* disrupts dynamic taint analysis, and *anti-alias analysis* obfuscates function calls by making them indirect.

Self-Modifying Code

Self-modification techniques dynamically alter portions of a program at runtime, rendering static analysis ineffective. This type of obfuscation forces the analyst into dynamic evaluation because the binary continuously modifies its own copy in memory during runtime.

Layered Obfuscation Transformations

Layered obfuscation transformations combine multiple obfuscation techniques. For our study, we combined all obfuscations pairwise and added two more complex layerings from the Tigress documentation (TigressRecipe1 and TigressRecipe2, https://tigress.wtf/recipes.html, accessed on 10 October 2025).

3.2. Dataset Creation and Feature Extraction

To systematically analyze the effects of source-level obfuscation in LLVM IR, we constructed a dataset that contains both a set of small, single-function C programs and a collection of more complex system utilities taken from the GNU Coreutils.

The first subset consists of 84 single-file C programs, each containing a single function. These programs implement a range of fundamental and well-known algorithms, such as for the calculation of cryptographic hashes, sorting, and compression. This collection provides a focused view on isolated obfuscation effects, as each sample is based on the same boilerplate code with minimal algorithm-specific code structures. About half of these programs are taken from the public obfuscation benchmark by Banescu et al. [18], while the remaining samples were written by us to specifically increase the variety of control-flow structures.

To complement the previously described minimal examples and to observe how obfuscation affects more complex software, we incorporated full-featured programs from the GNU Coreutils (version 9.5). These utilities such as `cat`, `sha256sum`, and `sort` represent real-world applications used in Linux distributions. Since Tigress operates on single-file C input, we first merged each Coreutils utility into a single C file using the Tigress `-merge` option. This step preserves the full functionality of the original tools while converting them into a form suitable for obfuscation and corresponding analysis.

All programs were then obfuscated using Tigress version 4.0.9. In total, a comprehensive set of around 70 transformation configurations, including both isolated and layered transformations, was applied to each sample. For layered transformations, all obfuscations were combined pairwise in both orders.

Following obfuscation, all source files were compiled to LLVM IR using `clang` with the `-S -emit-llvm` options. To account for compiler-influenced variations in the resulting IR, we used four optimization levels (O0 through O3) for each obfuscated version. The resulting IR files were grouped according to obfuscation configuration and compiler optimization level.

IR2Vec Embedding Generation

We then used IR2Vec [19] to convert LLVM IR components such as opcodes, types, and operands into vector embeddings. IR2Vec supports two encoding modes: symbolic, which considers IR structure, and flow-aware, which augments symbolic embeddings with control-flow information. For our experiments, we used the pre-trained IR2Vec model in symbolic (`sym`) mode with the default embedding dimensionality of 300. All compilation and embedding generation steps were executed within a Docker container based on Python 3.10. The container included LLVM/Clang and essential 32-bit libraries (`libc6:i386`, `libstdc++6:i386`, etc.) required for compiling C programs to LLVM IR. Embedding extraction was automated with Python scripts that compiled the C source

files to LLVM IR, processed them with IR2Vec to obtain the corresponding vector representations, and stored the resulting embeddings as structured CSV files for subsequent classifier training.

Dataset Limitations and Generalization

While our dataset provides a broad spectrum of obfuscation types, it is important to note that all transformations originate from a single obfuscation framework, namely Tigress [17]. This single-source origin may introduce a certain degree of bias, as the classifiers could, in part, learn Tigress-specific structural artifacts rather than entirely generalizable obfuscation patterns. Nevertheless, Tigress is the most comprehensive and widely used C to C obfuscator in academic research, offering a broad range of transformation types and configuration parameters [20,21]. Due to its configurability and frequent adoption in prior work, Tigress serves as a practical reference point for evaluating obfuscation analysis approaches. Future studies could expand on this dataset by integrating obfuscations produced by other frameworks to improve the generality of classification results.

3.3. The Final Dataset

The final dataset employed in this study comprises a diverse collection of obfuscated LLVM IR files, generated from multiple source programs transformed with a wide variety of obfuscation passes and combinations thereof, as mentioned in the previous section. A comprehensive breakdown of the dataset, split into a training and testing set, is presented in Table 1 (training set) and Table 2 (testing set), detailing class distributions and frequencies.

In addition to the obfuscated samples, we included a distinct class for completely *non-obfuscated*, i.e., non-processed, code.

Classes are balanced to make for fair comparison and performance assessment. Nevertheless, minor variations exist due to practical constraints encountered during dataset preparation (e.g., compilation failures or transformations not equally applicable to all programs). The training set constitutes approximately 80% of the entire dataset, while the remaining 20% forms a strictly isolated hold-out test set, ensuring that no information leakage occurs during the evaluation.

Table 1. Class distribution in the training set.

Class and Count		Class and Count		Class and Count	
EncodeArithmeticAntiTaintAnalysis	269	EncodeBranches	269	AntiTaintAnalysis	269
Virtualize	269	FlattenEncodeBranches	269	VirtualizeAntiTaintAnalysis	269
Flatten	269	EncodeArithmeticVirtualize	269	EncodeBranchesAntiTaintAnalysis	269
FlattenEncodeArithmetic	269	EncodeArithmeticEncodeBranches	269	AntiTaintAnalysisEncodeBranches	269
EncodeArithmeticFlatten	269	AntiTaintAnalysisFlatten	269	FlattenAntiTaintAnalysis	269
VirtualizeEncodeBranches	269	FlattenVirtualize	269	AntiTaintAnalysisVirtualize	269
VirtualizeFlatten	269	EncodeArithmeticEncodeLiterals	262	TigressRecipe2	262
EncodeLiteralsVirtualize	262	FlattenSelfModify	262	EncodeLiteralsFlatten	262
AntiAliasAnalysisFlatten	262	EncodeLiteralsEncodeBranches	262	VirtualizeAntiAliasAnalysis	262
EncodeLiteralsAntiTaintAnalysis	262	AntiAliasAnalysisEncodeLiterals	262	EncodeBranchesEncodeLiterals	262
AntiAliasAnalysisAntiTaintAnalysis	262	FlattenEncodeLiterals	262	VirtualizeEncodeLiterals	262
FlattenAntiAliasAnalysis	262	AntiAliasAnalysisVirtualize	262	EncodeLiterals	262
AntiAliasAnalysisEncodeBranches	262	AntiTaintAnalysisSplit	256	SplitEncodeArithmetic	256
EncodeBranchesSplit	256	EncodeBranchesEncodeArithmetic	256	EncodeArithmeticSplit	256
VirtualizeSplit	256	AntiTaintAnalysisEncodeArithmetic	256	Split	256
SplitAntiTaintAnalysis	256	SplitVirtualize	256	FlattenSplit	256
SplitFlatten	256	SplitEncodeBranches	256	EncodeArithmetic	256
VirtualizeEncodeArithmetic	256	AntiAliasAnalysis	250	EncodeArithmeticSelfModify	250
SelfModifyAntiAliasAnalysis	250	EncodeLiteralsEncodeArithmetic	250	EncodeArithmeticAntiAliasAnalysis	250
AntiAliasAnalysisEncodeArithmetic	250	TigressRecipe1	250	EncodeLiteralsAntiAliasAnalysis	250
EncodeBranchesAntiAliasAnalysis	250	EncodeLiteralsSplit	250	SplitAntiAliasAnalysis	250
AntiAliasAnalysisSplit	250	SplitEncodeLiterals	250	SplitSelfModify	249
SelfModifyEncodeArithmetic	249	SelfModify	249	non-obfuscated	67

Table 2. Class distribution in the testing set.

Class and Count		Class and Count		Class and Count	
Virtualize	67	EncodeArithmeticAntiTaintAnalysis	67	AntiTaintAnalysis	67
AntiTaintAnalysisEncodeBranches	67	FlattenAntiTaintAnalysis	67	FlattenVirtualize	67
FlattenEncodeArithmetic	67	FlattenEncodeBranches	67	EncodeBranchesAntiTaintAnalysis	67
VirtualizeAntiTaintAnalysis	67	AntiTaintAnalysisFlatten	67	VirtualizeEncodeBranches	67
EncodeBranches	67	EncodeArithmeticEncodeBranches	67	Flatten	67
VirtualizeFlatten	67	EncodeArithmeticVirtualize	67	EncodeArithmeticFlatten	67
AntiTaintAnalysisVirtualize	67	AntiAliasAnalysisVirtualize	66	FlattenSelfModify	66
VirtualizeEncodeLiterals	66	VirtualizeAntiAliasAnalysis	66	EncodeLiterals	66
EncodeLiteralsFlatten	66	AntiAliasAnalysisEncodeLiterals	66	FlattenEncodeLiterals	66
AntiAliasAnalysisFlatten	66	AntiAliasAnalysisEncodeBranches	66	EncodeArithmeticEncodeLiterals	66
EncodeLiteralsAntiTaintAnalysis	66	TigressRecipe2	66	FlattenAntiAliasAnalysis	66
EncodeLiteralsVirtualize	66	AntiAliasAnalysisAntiTaintAnalysis	66	EncodeBranchesEncodeLiterals	66
EncodeLiteralsEncodeBranches	66	SplitEncodeArithmetic	64	FlattenSplit	64
Split	64	AntiTaintAnalysisEncodeArithmetic	64	VirtualizeSplit	64
EncodeArithmetic	64	EncodeBranchesEncodeArithmetic	64	SplitEncodeBranches	64
SplitAntiTaintAnalysis	64	EncodeBranchesSplit	64	SplitVirtualize	64
SplitFlatten	64	EncodeArithmeticSplit	64	VirtualizeEncodeArithmetic	64
AntiTaintAnalysisSplit	64	SplitSelfModify	63	SelfModify	63
SelfModifyEncodeArithmetic	63	EncodeArithmeticAntiAliasAnalysis	62	EncodeLiteralsAntiAliasAnalysis	62
EncodeBranchesAntiAliasAnalysis	62	AntiAliasAnalysisEncodeArithmetic	62	EncodeArithmeticSelfModify	62
TigressRecipe1	62	AntiAliasAnalysisSplit	62	EncodeLiteralsEncodeArithmetic	62
EncodeLiteralsSplit	62	SplitEncodeLiterals	62	AntiAliasAnalysis	62
SplitAntiAliasAnalysis	62	SelfModifyAntiAliasAnalysis	62	non-obfuscated	17

3.4. Machine Learning

For our machine learning approach, we employed CatBoost, a modern boosting classifier [22,23], and ExtraTrees, an ensemble tree-based classifier known for its robust performance in classification tasks [24].

These boosting classifiers extend the basic concept of decision trees by sequentially training a series of weak learners, typically simple decision trees, and combining their predictions to form a strong, robust model. Unlike bagging techniques, which build multiple independent models in parallel, boosting methods iteratively focus on correcting the errors of their predecessors.

CatBoost distinguishes itself within this framework by incorporating a permutation-driven algorithm to handle categorical features natively, thereby reducing the need for extensive preprocessing. It is reported to achieve high accuracy and strong performance across diverse problem domains, even with the out-of-the-box implementation, without extensive hyperparameter optimization [25,26].

ExtraTrees (Extremely Randomized Trees) complements our approach by building multiple randomized decision trees on different subsets of the training data and aggregating their predictions to reduce variance and overfitting. ExtraTrees has shown excellent performance in tasks where robustness against noisy data or complex feature interactions is required [24]. Further, in past studies, ExtraTrees has been successfully used to identify if and how program code was obfuscated [10,11].

Evaluation Metrics

We report accuracy, precision, recall, and F1-score for each obfuscation class, providing several metrics to document our model's performance. Confusion matrices allow us to visualize misclassification patterns across different types (or layerings) of obfuscation.

3.5. Experimental Setup

Figure 1 illustrates the complete pipeline used in our experiments. We conducted our analysis separately using both CatBoost and ExtraTrees classifiers to evaluate their performance in classifying obfuscation types in LLVM IR.

Initially, the dataset was divided into a training set (80%) and a hold-out test set (20%) to evaluate model generalization.

To optimize our classifiers, we further split the training data into an 80/20 partition. We compared an out-of-the-box implementation of both CatBoost and ExtraTrees to a hyperparameter-optimized version using this split. We show this two-step split process in Figure 2.

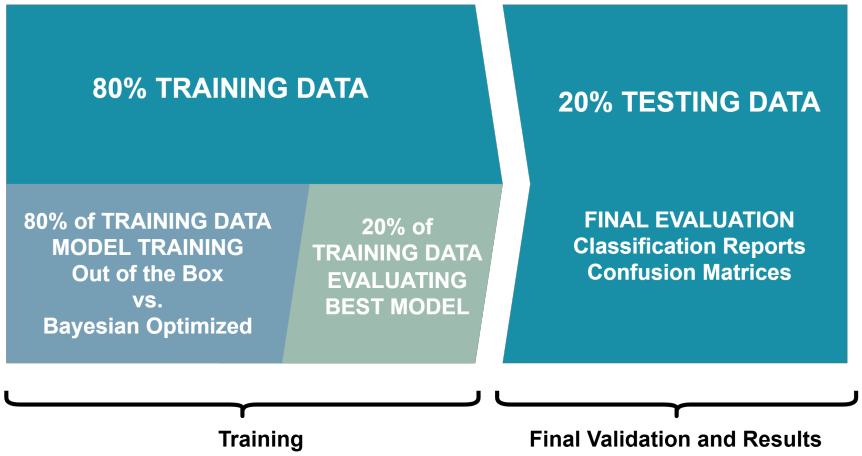

Figure 2. The employed two-step split process to train and validate our models.

It is worth noting that the out-of-the-box implementation of, e.g., CatBoost already performs very well on many datasets and even outperforms optimized quantum machine learning approaches on standard datasets, as discussed in [26]. Also, in our study, the Bayesian-optimized classifiers only slightly outperformed the out-of-the-box implementations, while the Bayesian optimization process is considerably more resource-intensive. For hyperparameter tuning, we employed Bayesian optimization with a 3-fold cross-validation strategy over 100 iterations on a predefined parameter grid [27]. Specifically, we used the `scikit-optimize` Python package [28] to search the hyperparameter space.

After identifying the optimal hyperparameters and model, we assessed generalization performance using the initially separated 20% hold-out test set. We computed standard evaluation metrics, including accuracy, precision, recall, and F1-score, and generated confusion matrices to gain deeper insights into each classifier's strengths and weaknesses.

Extended Cascading Experimental Design

In addition to the primary line of experiments, where both CatBoost and ExtraTrees were trained to directly classify all obfuscation types, we also developed a secondary, staged (cascading) research design to obtain a more fine-grained understanding of the classification behavior at multiple hierarchical levels. This approach extends the original setup while maintaining the same data splits and preprocessing pipeline. It allows us to analyze which stages of the obfuscation detection process are most reliable and which introduce the greatest uncertainty.

In this cascading design, we first create a single global stratified split of the cleaned dataset. From this base split, we derive several targeted experiments, each representing one level of the staged classification process. Each stage trains and evaluates its own classifier using the same feature representation and train/test division as in the primary setup.

The stages are defined as follows, and depicted in Figure 3:

- **Binary detection of obfuscation:** Determine whether a given sample is obfuscated or not.

- **Single vs. layered classification:** For all obfuscated samples, identify whether only one obfuscation method was applied or a combination of two or more obfuscations.
- **Single obfuscation identification:** Within the subset of single obfuscations, classify the specific obfuscation method used.
- **Layered obfuscation identification:** Within the subset of layered obfuscations, classify the obfuscations and the order in which they were applied to the code.
- **Optimization-level identification:** Identify the compiler optimization level (O-level).

Finally, we combine these stages to form a chained decision pipeline. In this integrated analysis, the classifier first determines whether a sample is obfuscated; if so, it predicts whether it is single or layered, and then classifies the specific obfuscation type within the predicted branch. This hierarchical inference simulates a decision process that incrementally narrows down the obfuscation characteristics of each sample.

We performed this cascading evaluation for both ExtraTrees and CatBoost classifiers. All classification reports and results for the ExtraTrees classifier are presented in the main text, while all results for the CatBoost classifier are presented in Appendix A.

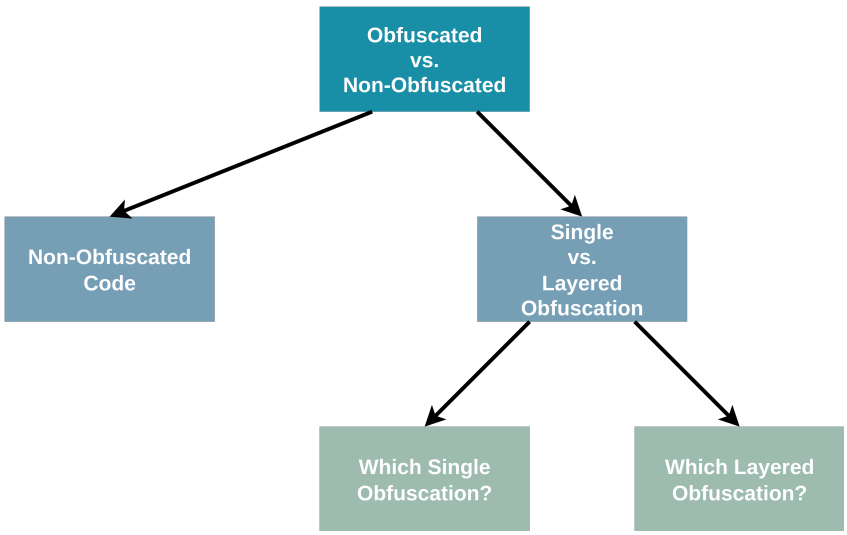

Figure 3. Overview of the extended cascading experimental design. Each node represents a classification stage, starting from binary obfuscation detection and progressing to finer-grained distinctions.

4. Results

This section presents the experimental outcomes across all conducted classification settings, following the staged structure introduced in Section 3.5. Each experiment evaluates a distinct prediction target, ranging from direct multiclass obfuscation identification to hierarchical cascading inference, and reports detailed metrics for the ExtraTrees models as the primary baseline. Corresponding CatBoost results are provided for comparison in Appendix A, maintaining identical evaluation protocols and presentation formats. For clarity and conciseness, all experiments are organized by level of abstraction: (i) direct multiclass prediction (Experiment 0), (ii) binary obfuscation detection (Experiment 1), (iii) single vs. layered discrimination (Experiment 2), (iv) single- and layered-obfuscation identification (Experiments 3–4), and (v) the integrated end-to-end cascade (Final Experiment).

Each subsection presents per-class classification reports, confusion matrix visualizations, and summary metrics on the hold-out test split. Extended per-class breakdowns for the complete obfuscation (O-level) classification are available in Appendix B, while the detailed CatBoost analogues mirroring all ExtraTrees experiments are consolidated in Appendix A. Together, these results provide an overview of the classifiers' performance across different levels of obfuscation granularity.

4.1. Multiclass Obfuscation Classification (Experiment 0)

Experiment 0 evaluates direct multiclass classification across all obfuscation labels without any prior filtering. We report results for the ExtraTrees model; the CatBoost counterpart is discussed in Appendix A (see Tables A1 and A2). The ExtraTrees classifier achieves an overall accuracy of **0.9312** on the hold-out test split. Per-class precision, recall, F1-scores, and supports are listed in Tables 3 and 4. Most single-pass transformations are identified with near-perfect scores. The main residual errors occur in the *EncodeLiterals*, *EncodeLiteralsAntiTaintAnalysis*, and some *Split*, *SelfModify* and *AntiAliasAnalysis* variants.

Table 3. Best ExtraTrees classifier, classification report with accuracy = 0.9312.

Obfuscation Type	Precision	Recall	F1-Score	Support	Obfuscation Type	Precision	Recall	F1-Score	Support
AntiAliasAnalysis	0.3294	0.4516	0.3810	62	AntiAliasAnalysisAntiTaintAnalysis	0.2093	0.1364	0.1651	66
AntiAliasAnalysisEncodeArithmetic	1.0000	1.0000	1.0000	62	AntiAliasAnalysisEncodeBranches	1.0000	1.0000	1.0000	66
AntiAliasAnalysisEncodeLiterals	1.0000	1.0000	1.0000	66	AntiAliasAnalysisFlatten	1.0000	1.0000	1.0000	66
AntiAliasAnalysisSplit	1.0000	1.0000	1.0000	62	AntiAliasAnalysisVirtualize	1.0000	1.0000	1.0000	66
AntiTaintAnalysis	1.0000	1.0000	1.0000	67	AntiTaintAnalysisEncodeArithmetic	0.9265	0.9844	0.9545	64
AntiTaintAnalysisEncodeBranches	1.0000	0.9851	0.9925	67	AntiTaintAnalysisFlatten	0.7792	0.8955	0.8333	67
AntiTaintAnalysisSplit	0.9412	1.0000	0.9697	64	AntiTaintAnalysisVirtualize	0.9565	0.9851	0.9706	67
EncodeArithmetic	1.0000	1.0000	1.0000	64	EncodeArithmeticAntiAliasAnalysis	1.0000	1.0000	1.0000	62
EncodeArithmeticAntiTaintAnalysis	0.9841	0.9254	0.9538	67	EncodeArithmeticEncodeBranches	0.9296	0.9851	0.9565	67
EncodeArithmeticEncodeLiterals	1.0000	1.0000	1.0000	66	EncodeArithmeticFlatten	0.9853	1.0000	0.9926	67
EncodeArithmeticSelfModify	1.0000	1.0000	1.0000	62	EncodeArithmeticSplit	0.9697	1.0000	0.9846	64
EncodeArithmeticVirtualize	1.0000	1.0000	1.0000	67	EncodeBranches	0.9710	1.0000	0.9853	67
EncodeBranchesAntiAliasAnalysis	1.0000	1.0000	1.0000	62	EncodeBranchesAntiTaintAnalysis	0.9853	1.0000	0.9926	67
EncodeBranchesEncodeArithmetic	0.9833	0.9219	0.9516	64	EncodeBranchesEncodeLiterals	1.0000	1.0000	1.0000	66
EncodeBranchesSplit	0.9552	1.0000	0.9771	64	EncodeLiterals	0.2750	0.3333	0.3014	66
EncodeLiteralsAntiAliasAnalysis	1.0000	1.0000	1.0000	62	EncodeLiteralsAntiTaintAnalysis	0.1538	0.1212	0.1356	66
EncodeLiteralsEncodeArithmetic	1.0000	1.0000	1.0000	62	EncodeLiteralsEncodeBranches	1.0000	1.0000	1.0000	66
EncodeLiteralsFlatten	1.0000	1.0000	1.0000	66	EncodeLiteralsSplit	1.0000	1.0000	1.0000	62
EncodeLiteralsVirtualize	1.0000	1.0000	1.0000	66	Flatten	1.0000	1.0000	1.0000	67
FlattenAntiAliasAnalysis	1.0000	1.0000	1.0000	66	FlattenAntiTaintAnalysis	0.8772	0.7463	0.8065	67
FlattenEncodeArithmetic	1.0000	0.9851	0.9925	67	FlattenEncodeBranches	1.0000	1.0000	1.0000	67
FlattenEncodeLiterals	1.0000	1.0000	1.0000	66	FlattenSelfModify	1.0000	1.0000	1.0000	66
FlattenSplit	1.0000	1.0000	1.0000	64	FlattenVirtualize	1.0000	1.0000	1.0000	67
SelfModify	0.7619	0.7619	0.7619	63	SelfModifyAntiAliasAnalysis	1.0000	1.0000	1.0000	62
SelfModifyEncodeArithmetic	0.7619	0.7619	0.7619	63	Split	0.7397	0.8438	0.7883	64
SplitAntiAliasAnalysis	1.0000	1.0000	1.0000	62	SplitAntiTaintAnalysis	1.0000	0.9375	0.9677	64
SplitEncodeArithmetic	0.8113	0.6719	0.7350	64	SplitEncodeBranches	1.0000	0.9531	0.9760	64
SplitEncodeLiterals	1.0000	1.0000	1.0000	62	SplitFlatten	1.0000	1.0000	1.0000	64
SplitSelfModify	1.0000	1.0000	1.0000	63	SplitVirtualize	1.0000	1.0000	1.0000	64
TigressRecipe1	1.0000	1.0000	1.0000	62	TigressRecipe2	1.0000	1.0000	1.0000	66
Virtualize	1.0000	1.0000	1.0000	67	VirtualizeAntiAliasAnalysis	1.0000	1.0000	1.0000	66
VirtualizeAntiTaintAnalysis	0.9846	0.9552	0.9697	67	VirtualizeEncodeArithmetic	1.0000	1.0000	1.0000	64
VirtualizeEncodeBranches	1.0000	1.0000	1.0000	67	VirtualizeEncodeLiterals	1.0000	1.0000	1.0000	66
VirtualizeFlatten	1.0000	1.0000	1.0000	67	VirtualizeSplit	1.0000	1.0000	1.0000	64
non-obfuscated	1.0000	0.8824	0.9375	17					

Table 4. Experiment 0 (ExtraTrees): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
ExtraTrees	0.9312	0.9315	0.9308	0.9304

Figure 4 presents representative sections of the confusion matrix highlighting the most common confusions: (i) *AntiAliasAnalysis* vs. *AntiAliasAnalysisAntiTaintAnalysis*, (ii) *EncodeLiterals* vs. *EncodeLiteralsAntiTaintAnalysis*, and (iii) *SelfModify* overlaps. These error clusters are consistent with later experiments, where literal encoding and layered combinations remain the primary sources of confusion.

Summary

Direct multiclass classification achieves stable performance with an overall accuracy of 93%. Most single-pass transformations are classified correctly, with F1-scores close to 1.0. The remaining errors are concentrated in (i) *EncodeLiterals* and its anti-taint variant, (ii) a subset of *Split* and *EncodeArithmetic* classes, and (iii) overlapping *SelfModify* variants. These error sources are consistent with the more complex layered classifications discussed in later sections.

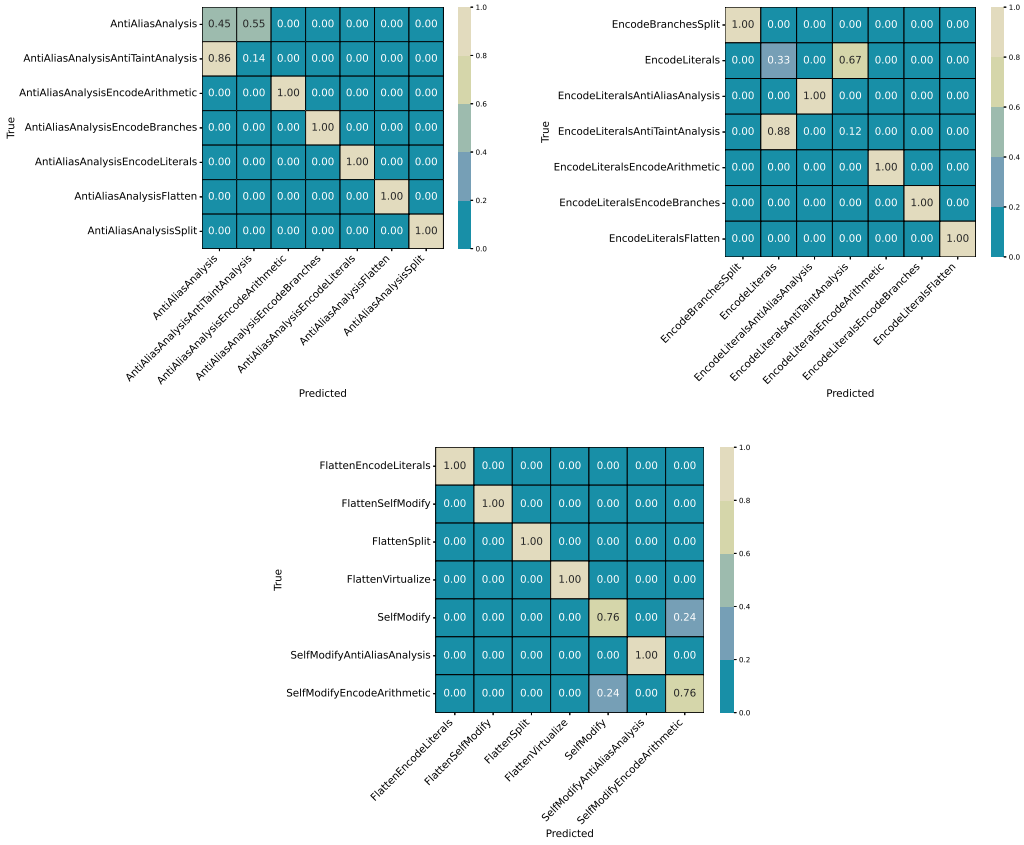

Figure 4. Experiment 0 (ExtraTrees): Confusion matrix excerpts (three snapshots presenting the most relevant confusions from the full relative matrix). **Top row**: *AntiAliasAnalysis* vs. *AntiAliasAnalysisAntiTaintAnalysis* (**left**) and *EncodeLiterals* cluster (**right**). **Bottom**: *SelfModify* cluster. Note that these confusion subsets do not fully represent the confusions across all classes. Some relative scores here are higher than in the whole picture, as some inter-class-confusions are not accounted for.

4.2. Cascading Experiments (ExtraTrees)

We evaluate a staged pipeline that mirrors a practical analysis workflow. Starting from the global stratified split (Section 3.5), we train one model per stage and route predictions sequentially:

- **Experiment 1 (Binary Obfuscation Detection):** *Obfuscated* vs. *Non_Obfuscated*. Results in Table 5 (per-class) and Table 6 (summary); confusion matrix in Figure 5.
- **Experiment 2 (Binary Single vs. Layered):** Conditioned on being obfuscated, classify *Single_Obf* vs. *Layered_Obf*. Results in Tables 7 and 8; confusion matrix in Figure 6.
- **Experiment 3 (Single-Obfuscation Identification, Multiclass):** Conditioned on *Single_Obf*, identify the specific method (nine classes). Results in Tables 9 and 10; confusion matrix in Figure 7.
- **Experiment 4 (Layered-Obfuscation Identification, Multiclass):** Conditioned on *Layered_Obf*, identify the layered label. Results in Tables 11 and 12.

Unless noted otherwise, we report metrics on the same hold-out test split as in Experiment 0 and use macro/weighted precision, recall, and F1 alongside overall accuracy. The CatBoost counterparts for these staged experiments are provided in Appendix A for completeness. The end-to-end performance of the integrated cascade is summarized in Section 4.3 (Table 13), with confusion-matrix snapshots in Figure 8.

4.2.1. Experiment 1: Binary Obfuscation Detection (Obfuscated vs. Non-Obfuscated)

The target variable is `ObfBinary` $\in$ {*Obfuscated*, *Non_Obfuscated*}. Table 5 reports per-class precision, recall, F1, and support for the ExtraTrees model on the hold-out test split. The corresponding summary metrics are listed in Table 6, and the confusion matrix is shown in Figure 5.

Table 5. Experiment 1 (ExtraTrees): Per-class report on the hold-out test split.

	Precision	Recall	F1-Score	Support
Non_Obfuscated	1.0000	0.7647	0.8667	17
Obfuscated	0.9991	1.0000	0.9995	4416

Table 6. Experiment 1 (ExtraTrees): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
ExtraTrees	0.9991	0.9996	0.8824	0.9331

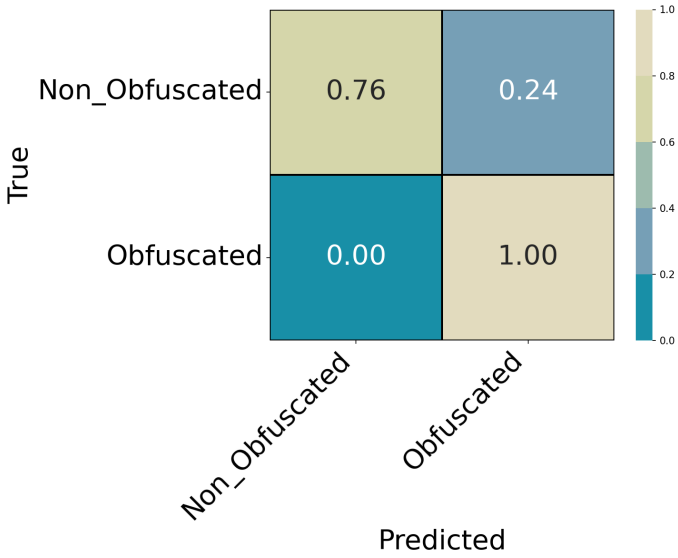

Figure 5. Experiment 1 (ExtraTrees): Confusion matrix for *Obfuscated* vs. *Non_Obfuscated* (relative).

Summary

Binary detection achieves an overall accuracy of **0.9991**. The *Obfuscated* class is predicted almost perfectly, while the smaller *Non_Obfuscated* class shows slightly lower recall (0.7647) due to limited sample size. These results confirm that the model reliably distinguishes obfuscated from clean samples. Consistent with later experiments, most remaining errors occur in subsequent stages of the cascade, while this initial detection step remains stable and highly accurate.

4.2.2. Experiment 2: Binary Single vs. Layered Obfuscations

This experiment operates on the subset of obfuscated samples only. The target variable is `SingleVsLayer` $\in$ {*Single_Obf*, *Layered_Obf*}. The task is to determine whether a given sample represents a single obfuscation or a layered combination of multiple transformations. Table 7 reports the per-class metrics, and Table 8 summarizes the overall performance. The confusion matrix of the best ExtraTrees model is shown in Figure 6.

Table 7. Experiment 2 (ExtraTrees): Per-class report on the hold-out test split.

	Precision	Recall	F1-Score	Support
Layered_Obf	0.9581	0.9974	0.9774	3829
Single_Obf	0.9767	0.7155	0.8260	587

Table 8. Experiment 2 (ExtraTrees): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
ExtraTrees	0.9599	0.9674	0.8564	0.9017

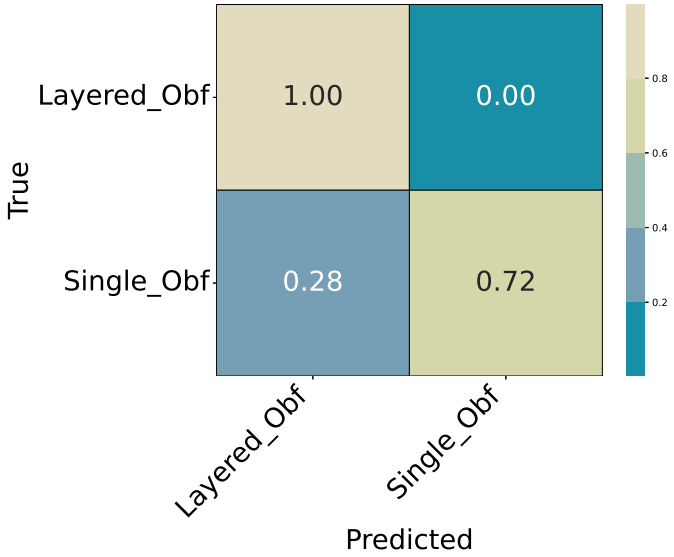

Figure 6. Experiment 2 (ExtraTrees): Confusion matrix for *Single_Obf* vs. *Layered_Obf* (relative).

Summary

The classifier achieves an overall accuracy of **0.9599**. The *Layered_Obf* class is identified with very high recall (0.9974), while the *Single_Obf* class shows a lower recall (0.7155), indicating that some single obfuscations are misclassified as layered. This asymmetry is visible in the confusion matrix (Figure 6), where most errors occur in that direction.

Overall, the classifier performs consistently across classes and provides a solid basis for the following staged experiments.

4.2.3. Experiment 3: Single-Obfuscation Identification (Multiclass)

This experiment isolates samples labeled as *Single_Obf* and identifies the specific obfuscation method applied. The target variable is `SingleMethod`, which includes nine transformations. The ExtraTrees classifier achieves a perfect accuracy of **1.0000** on the hold-out test split. Per-class precision, recall, and F1-scores are reported in Table 9, and the summary metrics are listed in Table 10. The corresponding confusion matrix is shown in Figure 7.

Table 9. Experiment 3 (ExtraTrees): Per-class report on the hold-out test split.

	Precision	Recall	F1-Score	Support
AntiAliasAnalysis	1.0000	1.0000	1.0000	62
AntiTaintAnalysis	1.0000	1.0000	1.0000	67
EncodeArithmetic	1.0000	1.0000	1.0000	64
EncodeBranches	1.0000	1.0000	1.0000	67
EncodeLiterals	1.0000	1.0000	1.0000	66
Flatten	1.0000	1.0000	1.0000	67
SelfModify	1.0000	1.0000	1.0000	63
Split	1.0000	1.0000	1.0000	64
Virtualize	1.0000	1.0000	1.0000	67

Table 10. Experiment 3 (ExtraTrees): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
ExtraTrees	1.0000	1.0000	1.0000	1.0000

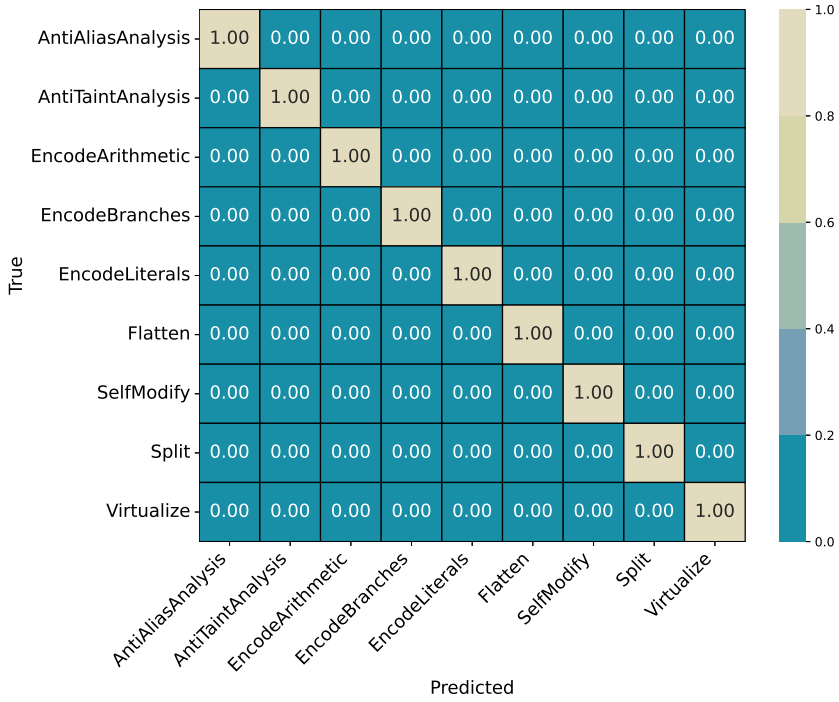

Figure 7. Experiment 3 (ExtraTrees): Confusion matrix over all single obfuscation classes (relative).

Summary

The classifier perfectly distinguishes all nine single obfuscation methods without any misclassifications. This confirms that the IR2Vec feature representation preserves clear structural and semantic differences between individual transformations. The confusion matrix (Figure 7) shows complete diagonal dominance, indicating that each obfuscation type forms a distinct region in the feature space. At this stage, separability among individual transformations is fully achieved. Later experiments focus on more complex scenarios, such as layered obfuscations and O-level identification, where overlaps between transformations are more likely.

4.2.4. Experiment 4: Layered-Obfuscation Identification (Multiclass)

This multiclass experiment uses only samples labeled as *Layered_Obf* and identifies the specific combination of transformations applied. The target variable `LayeredLabel`

encodes each unique layered configuration. The ExtraTrees classifier achieves an overall accuracy of **0.9867** on the hold-out test split. Per-class precision, recall, and F1-scores are reported in Table 11, while summary metrics are listed in Table 12.

Table 11. Experiment 4, only layered Obfuscations (ExtraTrees): Per-class report on the hold-out test split.

Class	Precision	Recall	F1-Score	Support	Class	Precision	Recall	F1-Score	Support
AntiAliasAnalysisAntiTaintAnalysis	1.0000	1.0000	1.0000	66	EncodeLiteralsAntiAliasAnalysis	1.0000	1.0000	1.0000	62
AntiAliasAnalysisEncodeArithmetic	1.0000	1.0000	1.0000	62	EncodeLiteralsAntiTaintAnalysis	1.0000	1.0000	1.0000	66
AntiAliasAnalysisEncodeBranches	1.0000	1.0000	1.0000	66	EncodeLiteralsEncodeArithmetic	1.0000	1.0000	1.0000	62
AntiAliasAnalysisEncodeLiterals	1.0000	1.0000	1.0000	66	EncodeLiteralsEncodeBranches	1.0000	1.0000	1.0000	66
AntiAliasAnalysisFlatten	1.0000	1.0000	1.0000	66	EncodeLiteralsFlatten	1.0000	1.0000	1.0000	66
AntiAliasAnalysisSplit	1.0000	1.0000	1.0000	62	EncodeLiteralsSplit	1.0000	1.0000	1.0000	62
AntiAliasAnalysisVirtualize	1.0000	1.0000	1.0000	66	EncodeLiteralsVirtualize	1.0000	1.0000	1.0000	66
AntiTaintAnalysisEncodeArithmetic	0.9265	0.9844	0.9545	64	FlattenAntiAliasAnalysis	1.0000	1.0000	1.0000	66
AntiTaintAnalysisEncodeBranches	1.0000	0.9851	0.9925	67	FlattenAntiTaintAnalysis	0.8772	0.7463	0.8065	67
AntiTaintAnalysisFlatten	0.7792	0.8955	0.8333	67	FlattenEncodeArithmetic	1.0000	0.9851	0.9925	67
AntiTaintAnalysisSplit	0.9412	1.0000	0.9697	64	FlattenEncodeBranches	1.0000	1.0000	1.0000	67
AntiTaintAnalysisVirtualize	0.9565	0.9851	0.9706	67	FlattenEncodeLiterals	1.0000	1.0000	1.0000	66
EncodeArithmeticAntiAliasAnalysis	1.0000	1.0000	1.0000	62	FlattenSelfModify	1.0000	1.0000	1.0000	66
EncodeArithmeticAntiTaintAnalysis	0.9841	0.9254	0.9538	67	FlattenSplit	1.0000	1.0000	1.0000	64
EncodeArithmeticEncodeBranches	0.9296	0.9851	0.9565	67	FlattenVirtualize	1.0000	1.0000	1.0000	67
EncodeArithmeticEncodeLiterals	1.0000	1.0000	1.0000	66	SelfModifyAntiAliasAnalysis	1.0000	1.0000	1.0000	62
EncodeArithmeticFlatten	0.9853	1.0000	0.9926	67	SelfModifyEncodeArithmetic	1.0000	1.0000	1.0000	63
EncodeArithmeticSelfModify	1.0000	1.0000	1.0000	62	SplitAntiAliasAnalysis	1.0000	1.0000	1.0000	62
EncodeArithmeticSplit	0.9697	1.0000	0.9846	64	SplitAntiTaintAnalysis	1.0000	0.9375	0.9677	64
EncodeArithmeticVirtualize	1.0000	1.0000	1.0000	67	SplitEncodeArithmetic	1.0000	0.9688	0.9841	64
EncodeBranchesAntiAliasAnalysis	1.0000	1.0000	1.0000	62	SplitEncodeBranches	1.0000	0.9531	0.9760	64
EncodeBranchesAntiTaintAnalysis	0.9853	1.0000	0.9926	67	SplitEncodeLiterals	1.0000	1.0000	1.0000	62
EncodeBranchesEncodeArithmetic	0.9833	0.9219	0.9516	64	SplitFlatten	1.0000	1.0000	1.0000	64
EncodeBranchesEncodeLiterals	1.0000	1.0000	1.0000	66	SplitSelfModify	1.0000	1.0000	1.0000	63
EncodeBranchesSplit	0.9552	1.0000	0.9771	64	SplitVirtualize	1.0000	1.0000	1.0000	64
TigressRecipe1	1.0000	1.0000	1.0000	62	TigressRecipe2	1.0000	1.0000	1.0000	66
VirtualizeAntiAliasAnalysis	1.0000	1.0000	1.0000	66	VirtualizeAntiTaintAnalysis	0.9846	0.9552	0.9697	67
VirtualizeEncodeArithmetic	1.0000	1.0000	1.0000	64	VirtualizeEncodeBranches	1.0000	1.0000	1.0000	67
VirtualizeEncodeLiterals	1.0000	1.0000	1.0000	66	VirtualizeFlatten	1.0000	1.0000	1.0000	67
VirtualizeSplit	1.0000	1.0000	1.0000	64					

Table 12. Experiment 4 (ExtraTrees): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
ExtraTrees	0.9867	0.9874	0.9869	0.9869

Summary

The classifier achieves an overall accuracy of **0.9867**, with near-uniform performance across all layered configurations. Most classes are predicted without error, while a few closely related combinations, such as *FlattenAntiTaintAnalysis* and *EncodeArithmeticAntiTaintAnalysis*, show minor confusion. This result demonstrates that the IR2Vec features and the classifier remain effective even under complex multi-stage transformation scenarios.

4.3. Final Cascaded Inference (Integrated Pipeline, ExtraTrees)

We evaluate the full cascaded pipeline on the hold-out test split. The chain applies stage models in sequence: obfuscation detection (Experiment 1), single vs. layered classification (Experiment 2) for obfuscated samples, and method identification using the appropriate model (Experiment 3 for single obfuscations; Experiment 4 for layered ones). Table 13 reports per-class precision/recall/F1 and supports for the final predicted label, overall metrics are shown in Table 14, i.e., the end-to-end accuracy is **0.9477**. Figure 8 shows three snapshots from the final confusion matrix. .

Table 13. Final cascading classification report.

Class	Precision	Recall	F1-Score	Support	Class	Precision	Recall	F1-Score	Support
AntiAliasAnalysis	0.0000	0.0000	0.0000	62	EncodeLiteralsAntiAliasAnalysis	1.0000	1.0000	1.0000	62
AntiAliasAnalysisAntiTaintAnalysis	0.5156	1.0000	0.6804	66	EncodeLiteralsAntiTaintAnalysis	0.5000	1.0000	0.6667	66
AntiAliasAnalysisEncodeArithmetic	1.0000	1.0000	1.0000	62	EncodeLiteralsEncodeArithmetic	1.0000	1.0000	1.0000	62
AntiAliasAnalysisEncodeBranches	1.0000	1.0000	1.0000	66	EncodeLiteralsEncodeBranches	1.0000	1.0000	1.0000	66
AntiAliasAnalysisEncodeLiterals	1.0000	1.0000	1.0000	66	EncodeLiteralsFlatten	1.0000	1.0000	1.0000	66
AntiAliasAnalysisFlatten	1.0000	1.0000	1.0000	66	EncodeLiteralsSplit	1.0000	1.0000	1.0000	62
AntiAliasAnalysisSplit	1.0000	1.0000	1.0000	62	EncodeLiteralsVirtualize	1.0000	1.0000	1.0000	66
AntiAliasAnalysisVirtualize	1.0000	1.0000	1.0000	66	FlattenAntiAliasAnalysis	1.0000	1.0000	1.0000	66
AntiTaintAnalysis	1.0000	0.9701	0.9848	67	FlattenAntiTaintAnalysis	0.8772	0.7463	0.8065	67
AntiTaintAnalysisEncodeArithmetic	0.9265	0.9844	0.9545	64	FlattenEncodeArithmetic	1.0000	0.9851	0.9925	67
AntiTaintAnalysisEncodeBranches	0.9706	0.9851	0.9778	67	FlattenEncodeBranches	1.0000	1.0000	1.0000	67
AntiTaintAnalysisFlatten	0.7792	0.8955	0.8333	67	FlattenEncodeLiterals	1.0000	1.0000	1.0000	66
AntiTaintAnalysisSplit	0.9412	1.0000	0.9697	64	FlattenSelfModify	1.0000	1.0000	1.0000	66
AntiTaintAnalysisVirtualize	0.9565	0.9851	0.9706	67	FlattenSplit	1.0000	1.0000	1.0000	64
EncodeArithmetic	0.9846	1.0000	0.9922	64	FlattenVirtualize	1.0000	1.0000	1.0000	67
EncodeArithmeticAntiAliasAnalysis	1.0000	1.0000	1.0000	62	SelfModify	0.8654	0.7143	0.7826	63
EncodeArithmeticAntiTaintAnalysis	0.9841	0.9254	0.9538	67	SelfModifyAntiAliasAnalysis	1.0000	1.0000	1.0000	62
EncodeArithmeticEncodeBranches	0.9296	0.9851	0.9565	67	SelfModifyEncodeArithmetic	0.7568	0.8889	0.8175	63
EncodeArithmeticEncodeLiterals	1.0000	1.0000	1.0000	66	Split	0.9375	0.7031	0.8036	64
EncodeArithmeticFlatten	0.9853	1.0000	0.9926	67	SplitAntiAliasAnalysis	1.0000	1.0000	1.0000	62
EncodeArithmeticSelfModify	1.0000	1.0000	1.0000	62	SplitAntiTaintAnalysis	1.0000	0.9375	0.9677	64
EncodeArithmeticSplit	0.9697	1.0000	0.9846	64	SplitEncodeArithmetic	0.7867	0.9219	0.8489	64
EncodeArithmeticVirtualize	1.0000	1.0000	1.0000	67	SplitEncodeBranches	0.9683	0.9531	0.9606	64
EncodeBranches	0.9571	1.0000	0.9781	67	SplitEncodeLiterals	1.0000	1.0000	1.0000	62
EncodeBranchesAntiAliasAnalysis	1.0000	1.0000	1.0000	62	SplitFlatten	1.0000	1.0000	1.0000	64
EncodeBranchesAntiTaintAnalysis	0.9853	1.0000	0.9926	67	SplitSelfModify	1.0000	1.0000	1.0000	63
EncodeBranchesEncodeArithmetic	0.9833	0.9219	0.9516	64	SplitVirtualize	1.0000	1.0000	1.0000	64
EncodeBranchesEncodeLiterals	1.0000	1.0000	1.0000	66	TigressRecipe1	1.0000	1.0000	1.0000	62
EncodeBranchesSplit	0.9412	1.0000	0.9697	64	TigressRecipe2	1.0000	1.0000	1.0000	66
EncodeLiterals	0.0000	0.0000	0.0000	66	Virtualize	1.0000	1.0000	1.0000	67
Flatten	1.0000	1.0000	1.0000	67	VirtualizeAntiAliasAnalysis	1.0000	1.0000	1.0000	66
VirtualizeAntiTaintAnalysis	0.9846	0.9552	0.9697	67	VirtualizeEncodeArithmetic	1.0000	1.0000	1.0000	64
VirtualizeEncodeBranches	1.0000	1.0000	1.0000	67	VirtualizeEncodeLiterals	1.0000	1.0000	1.0000	66
VirtualizeFlatten	1.0000	1.0000	1.0000	67	VirtualizeSplit	1.0000	1.0000	1.0000	64
non-obfuscated	1.0000	0.7647	0.8667	17					

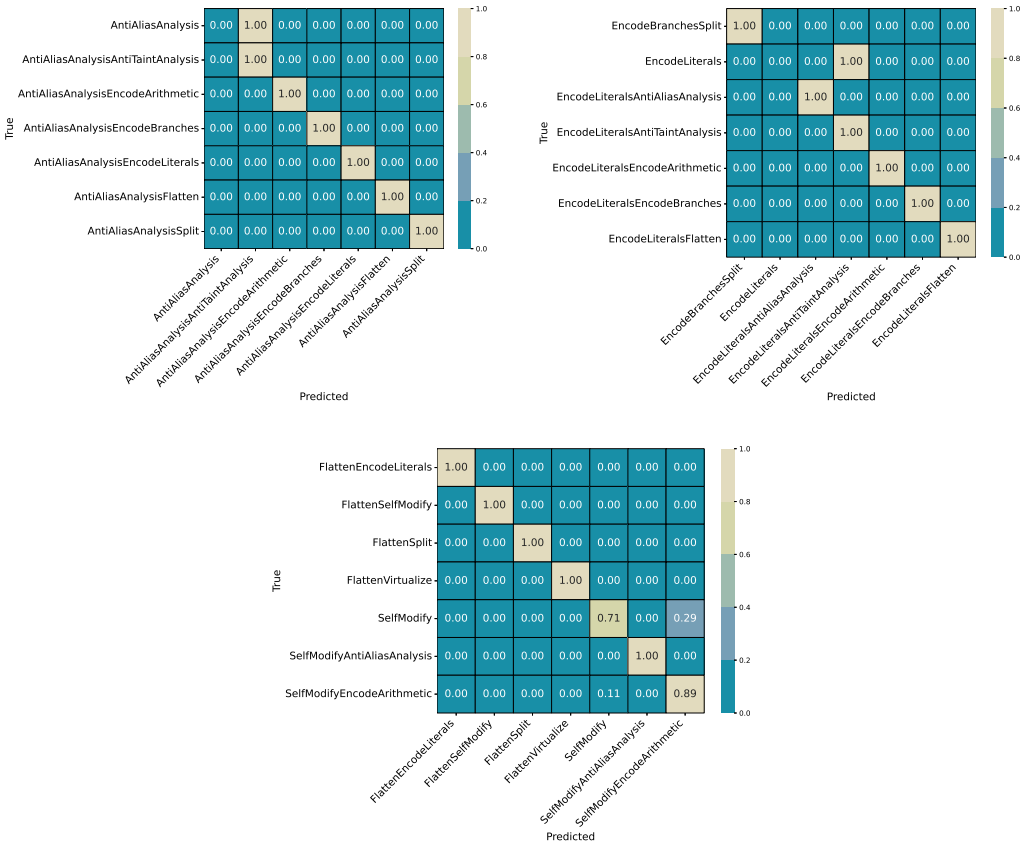

Figure 8. Final cascaded inference (ExtraTrees): selected confusion matrix regions showing the most relevant end-to-end misclassifications in the integrated pipeline. **Top left**: confusion between *AntiAliasAnalysis* and its layered variant *AntiAliasAnalysisAntiTaintAnalysis*. **Top right**: *EncodeLiterals*

family with misclassifications propagated from the single-versus-layered decision stage. **Bottom**: confusion cluster in the *SelfModify* and *Split* families. These regions illustrate how localized stage errors propagate through the cascade, while overall class separability remains high. Note that these confusion subsets do not fully represent the confusions across all classes. Some relative scores here are higher than in the whole picture, as some inter-class-confusions are not accounted for.

Table 14. Experiment 4 (ExtraTrees): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
ExtraTrees	0.9477	0.9346	0.9453	0.9366

Summary

The cascaded pipeline reaches **94.77%** accuracy. Most layered and many single-pass classes remain highly accurate end-to-end. Persistent errors align with earlier observations: base *EncodeLiterals* and *AntiAliasAnalysis* show low recall in the final output (both are challenging upstream), while *Non_Obfuscated* retains the same recall as in Experiment 1 (0.7647) due to its small support. Overall macro/weighted metrics (**Macro F1 = 0.9366**, **Weighted F1 = 0.9377**) are consistent with the stage-wise results, confirming that the pipeline composes well in practice.

5. Discussion

Across all stages, both classifiers, ExtraTrees and CatBoost, perform strongly on IR2Vec embeddings of Tigress–obfuscated LLVM IR. The staged/cascaded evaluations show that most errors appear in a few predictable areas, mainly in base *EncodeLiterals*, *AntiAliasAnalysis*, and some *Split*/*SelfModify* variants, while the remaining labels are classified reliably.

ExtraTrees (ET) performs slightly better than CatBoost (CB) in most settings, usually by about one to two percentage points. In the flat multiclass task (Experiment 0), ET reaches an accuracy of **0.9312** compared to **0.9269** for CB. Confusions occur in similar places for both, such as between *EncodeLiterals*, *EncodeLiteralsAntiTaintAnalysis*, and certain *Split*/*SelfModify* combinations. In binary obfuscation detection (Experiment 1), both models are nearly perfect with accuracies of **0.9991** for ET and **0.9993** for CB. The only consistent weakness is recall on *Non_Obfuscated* samples, mainly due to the smaller number of clean examples.

In the binary classification between *Single_Obf* and *Layered_Obf* (Experiment 2), ET reaches **0.9599** accuracy, and CB **0.9522**. Both models show lower recall for *Single_Obf* (ET **0.7155**, CB **0.6729**), as some single transformations, such as *Flatten* or *EncodeLiterals*, resemble layered obfuscations.

For identifying single obfuscation methods (Experiment 3), both classifiers reach **1.0000** across all nine transformations. The embeddings preserve clear distinctions between transformation types, confirming that IR2Vec captures characteristic structural patterns. In the layered-obfuscation task (Experiment 4), both models maintain high performance (ET **0.9867**, CB **0.9851**). Small confusions appear in combinations involving *AntiTaintAnalysis*, *Flatten*, or *EncodeArithmetic*.

The decision pipeline integrates several individually trained models, each specialized for a distinct decision stage. The first stage determines whether a sample is obfuscated at all, effectively separating clean IR from transformed IR. The second stage, trained only on obfuscated samples, decides whether the code was modified by a single transformation or by a layered combination. The third and fourth stages then specialize on their respective domains: one model identifies the specific single obfuscation method, while the other identifies the layered combination. This modular setup allows the system to narrow down uncertainty step by step, reducing the complexity of each decision. The output from one stage serves as the input domain for the next, creating a structured inference chain that

resembles a multi-step diagnostic process. Although this design introduces the possibility of error propagation between stages, it improves interpretability and isolates where specific classification difficulties arise. In practice, nearly all residual misclassifications in the final output can be traced to errors introduced in one of the intermediate stages, particularly in the single-versus-layered split.

The final cascaded pipeline combines all stages into an integrated inference chain. Here, ET reaches an overall accuracy of **0.9477** (Macro F1 **0.9366**), while CB reaches **0.9276** (Macro F1 **0.9179**). The general accuracy difference remains consistent, but the cascaded design highlights where specific misclassifications originate. The main residual issues correspond to upstream weaknesses:

- Base *EncodeLiterals* still shows very low recall for both classifiers due to its overlap with layered variants.
- The pair *AntiAliasAnalysis* and *AntiAliasAnalysisAntiTaintAnalysis* remains difficult. ET achieves 100% recall for the layered form but misses all base samples, while CB shows similar behavior with slightly lower precision and recall.
- For *Virtualize* and *VirtualizeAntiTaintAnalysis*, CB loses some recall (around 0.85–0.87), whereas ET remains stable.
- *Split* and *SelfModify* also degrade slightly in the cascaded setup compared to their isolated performance.

Each stage contributes differently to the final accuracy. The first classifier, which detects obfuscation, filters the input almost perfectly and prevents contamination from clean samples. The second classifier is the most critical, as errors here directly propagate and determine whether a sample is sent to the correct downstream model. Its 95% accuracy ensures stable routing but still accounts for most of the residual confusion in the cascade. The third and fourth classifiers perform with near-perfect precision within their subdomains, showing that once a sample is correctly routed, the specialized models can identify transformation signatures almost without error. Together, these results demonstrate that most of the cascading loss stems from misrouting rather than model uncertainty within each stage.

The cascaded model slgihtly exceeds the flat multiclass accuracy but keeps performance high while making the model's behavior easier to interpret. Because the binary obfuscation stage (Experiment 1) and single-method classification (Experiment 3) are nearly perfect, most end-to-end losses originate from the single-versus-layered distinction (Experiment 2) and a few difficult layered combinations in Experiment 4.

Across all experiments, the main trends remain stable. IR2Vec embeddings are highly discriminative at the method level and robust for layered transformations. Remaining errors occur in the same few areas across both classifiers and are mainly due to class imbalance and overlapping transformation effects. ET performs slightly better in Experiments 0, 2, 4, and in the final cascade, while CB matches or slightly exceeds ET in Experiment 1 and performs equally in Experiment 3.

In summary, IR-level obfuscation classification with IR2Vec embeddings achieves strong and consistent results across both classifiers. The cascaded setup preserves accuracy (ET **0.9477**, CB **0.9276**) comparable to the flat baseline (ET **0.9312**, CB **0.9269**) while making stage-specific error sources visible. Adding targeted post hoc checks for known overlaps, such as between *EncodeLiterals* and its layered variants or between *AntiAliasAnalysis* and *AntiAliasAnalysisAntiTaintAnalysis*, could reduce residual ambiguity without changing the embedding or classifier setup.

6. Conclusion and Future Work

This study showed that IR2Vec embeddings combined with ensemble classifiers, specifically ExtraTrees and CatBoost, can effectively identify a broad range of obfuscation transformations. Both models achieved consistently high accuracy across all experimental stages, demonstrating that structural and semantic information in IR2Vec embeddings is sufficient for reliable obfuscation detection. The results confirm that even simple machine learning models can accurately separate obfuscated from non-obfuscated code, distinguish single from layered transformations, and identify individual obfuscation methods with high precision.

Across all experiments, ExtraTrees achieved slightly higher accuracy than CatBoost, typically by one to two percentage points, while CatBoost matched or exceeded ExtraTrees in binary detection tasks. The cascaded decision pipeline, which combines several specialized classifiers trained for distinct sub-tasks, reached an overall accuracy of **0.9477** for ExtraTrees and **0.9276** for CatBoost. This design allows classification uncertainty to be isolated and traced to individual decision stages. Errors primarily originate in the single-versus-layered classification step, while the preceding obfuscation detector and downstream specialized classifiers perform nearly perfectly. The cascading approach therefore improves interpretability without compromising accuracy, providing insight into how uncertainty propagates through the decision process.

Single obfuscation methods were identified with perfect accuracy by both classifiers, confirming that IR2Vec embeddings capture unique structural and semantic signatures for each transformation. Layered combinations were also recognized with high precision, though certain overlaps, particularly between *AntiAliasAnalysis* and *AntiTaintAnalysis*, or between literal and arithmetic encodings, remain challenging. These confusions stem from structural similarity in the intermediate representation. Still, even in these cases, classification performance remains above 95% accuracy, showing the robustness of the embedding–classifier combination.

While the flat multiclass setup already performs strongly (ET **0.9312**, CB **0.9269**), the staged design offers a clear analytical advantage by decomposing complex classification decisions into smaller, interpretable subproblems. It provides a transparent view of how individual classification steps interact and where errors originate, making it more suitable for applied analysis scenarios where reliability and explainability are as important as raw accuracy.

Since intermediate representations can be reconstructed from compiled binaries, applying this workflow to lifted IR from real-world executables would broaden its practical applicability in future work and enable automated classification of obfuscation directly from binary code.

All machine learning experiments and data are presented in a corresponding repository, https://github.com/Raubkatz/IR2Vec_Obfuscation_Identification (accessed on 10 October 2025).

Author Contributions: Conceptualization, S.R. and S.S.; methodology, S.R., S.S., and P.F.; software, S.R., S.S., and P.F.; validation, S.R., S.S., and P.F.; formal analysis, S.R., P.F., and S.S.; investigation, S.R., S.S., and P.F.; resources, S.R. and K.M.; data curation, S.R., P.F., and S.S.; writing—original draft preparation, S.R., S.S., P.F., and K.M.; writing—review and editing, S.R., P.F., S.S., and K.M.; visualization, S.R.; supervision, S.R., S.S., and K.M.; project administration, S.S. and K.M.; funding acquisition, S.S. and K.M. All authors have read and agreed to the published version of the manuscript.

Funding: The financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged. SBA Research (SBA-K1 NGC) is a COMET Center within the COMET—Competence Centers for Excellent Technologies Programme and funded by BMIMI, BMWET, and the federal state of Vienna. The COMET Programme is managed by FFG.

This research was in parts funded by the Austrian Science Fund (FWF): I3646-N31. The authors acknowledge the funding by the University of Vienna for financial support through its Open Access Funding Program.

Data Availability Statement: The dataset used for all machine learning experiments is part of the corresponding GitHub Repository (https://github.com/) at https://github.com/Raubkatz/IR2Vec_Obfuscation_Identification, accessed on 10 October 2025.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. CatBoost Results for Cascading Experiments

Appendix A.1. Multiclass Obfuscation Classification (Experiment 0)

Experiment 0 evaluates direct multiclass classification across all obfuscation labels without prior filtering. Here we report the CatBoost model; ExtraTrees results are provided in the main text (Section 4) for comparison. CatBoost attains an overall accuracy of **0.9269** on the hold-out test split. Per-class precision, recall, F1, and supports are summarized in Tables A1 and A2, corresponding confusion matrix excerpts are given in Figure A1. As with ExtraTrees, most single-pass and many layered classes are near-perfect. The main errors remain in *EncodeLiterals* and related variants, *AntiAliasAnalysis*, and a subset of *Split*. Compared with ExtraTrees, CatBoost shows slightly lower recall for *VirtualizeAntiTaintAnalysis* and slightly higher recall for *non-obfuscated*.

Summary

CatBoost achieves an overall accuracy of **92.69%**. Most single-pass and many-layered classes are predicted correctly (F1 near 1.0). Errors concentrate in *EncodeLiterals* and *EncodeLiteralsAntiTaintAnalysis*, *AntiAliasAnalysis*, and selected *Split* cases. Compared with ExtraTrees, CatBoost shows lower recall for *VirtualizeAntiTaintAnalysis* (0.8806) and higher recall for *non-obfuscated* (0.9412).

Table A1. Best CatBoost classifier, per-class report.

Class	Precision	Recall	F1-Score	Support	Class	Precision	Recall	F1-Score	Support
AntiAliasAnalysis	0.2623	0.2581	0.2602	62	AntiAliasAnalysisAntiTaintAnalysis	0.3134	0.3182	0.3158	66
AntiAliasAnalysisEncodeArithmetic	1.0000	1.0000	1.0000	62	AntiAliasAnalysisEncodeBranches	1.0000	1.0000	1.0000	66
AntiAliasAnalysisEncodeLiterals	1.0000	1.0000	1.0000	66	AntiAliasAnalysisFlatten	1.0000	1.0000	1.0000	66
AntiAliasAnalysisSplit	1.0000	1.0000	1.0000	62	AntiAliasAnalysisVirtualize	1.0000	1.0000	1.0000	66
AntiTaintAnalysis	1.0000	1.0000	1.0000	67	AntiTaintAnalysisEncodeArithmetic	0.9265	0.9844	0.9545	64
AntiTaintAnalysisEncodeBranches	0.9545	0.9403	0.9474	67	AntiTaintAnalysisFlatten	0.8028	0.8507	0.8261	67
AntiTaintAnalysisSplit	0.8649	1.0000	0.9275	64	AntiTaintAnalysisVirtualize	0.8919	0.9851	0.9362	67
EncodeArithmetic	1.0000	1.0000	1.0000	64	EncodeArithmeticAntiAliasAnalysis	1.0000	1.0000	1.0000	62
EncodeArithmeticAntiTaintAnalysis	0.9841	0.9254	0.9538	67	EncodeArithmeticEncodeBranches	0.9296	0.9851	0.9565	67
EncodeArithmeticEncodeLiterals	1.0000	1.0000	1.0000	66	EncodeArithmeticFlatten	0.9853	1.0000	0.9926	67
EncodeArithmeticSelfModify	1.0000	1.0000	1.0000	62	EncodeArithmeticSplit	0.9697	1.0000	0.9846	64
EncodeArithmeticVirtualize	1.0000	1.0000	1.0000	67	EncodeBranches	0.9853	1.0000	0.9926	67
EncodeBranchesAntiAliasAnalysis	1.0000	1.0000	1.0000	62	EncodeBranchesAntiTaintAnalysis	0.9412	0.9552	0.9481	67
EncodeBranchesEncodeArithmetic	0.9833	0.9219	0.9516	64	EncodeBranchesEncodeLiterals	1.0000	1.0000	1.0000	66
EncodeBranchesSplit	0.9692	0.9844	0.9767	64	EncodeLiterals	0.2273	0.2273	0.2273	66
EncodeLiteralsAntiAliasAnalysis	1.0000	1.0000	1.0000	62	EncodeLiteralsAntiTaintAnalysis	0.2273	0.2273	0.2273	66
EncodeLiteralsEncodeArithmetic	1.0000	1.0000	1.0000	62	EncodeLiteralsEncodeBranches	1.0000	1.0000	1.0000	66
EncodeLiteralsFlatten	1.0000	1.0000	1.0000	66	EncodeLiteralsSplit	1.0000	1.0000	1.0000	62
EncodeLiteralsVirtualize	1.0000	1.0000	1.0000	66	Flatten	1.0000	1.0000	1.0000	67
FlattenAntiAliasAnalysis	1.0000	1.0000	1.0000	66	FlattenAntiTaintAnalysis	0.8413	0.7910	0.8154	67
FlattenEncodeArithmetic	1.0000	0.9851	0.9925	67	FlattenEncodeBranches	1.0000	1.0000	1.0000	67
FlattenEncodeLiterals	1.0000	1.0000	1.0000	66	FlattenSelfModify	1.0000	1.0000	1.0000	66
FlattenSplit	1.0000	1.0000	1.0000	64	FlattenVirtualize	1.0000	1.0000	1.0000	67
SelfModify	0.7619	0.7619	0.7619	63	SelfModifyAntiAliasAnalysis	1.0000	1.0000	1.0000	62
SelfModifyEncodeArithmetic	0.7619	0.7619	0.7619	63	Split	0.7692	0.7812	0.7752	64
SplitAntiAliasAnalysis	1.0000	1.0000	1.0000	62	SplitAntiTaintAnalysis	1.0000	0.8438	0.9153	64
SplitEncodeArithmetic	0.7705	0.7344	0.7520	64	SplitEncodeBranches	0.9841	0.9688	0.9764	64
SplitEncodeLiterals	1.0000	1.0000	1.0000	62	SplitFlatten	1.0000	1.0000	1.0000	64
SplitSelfModify	1.0000	1.0000	1.0000	63	SplitVirtualize	1.0000	1.0000	1.0000	64
TigressRecipe1	1.0000	1.0000	1.0000	62	TigressRecipe2	1.0000	1.0000	1.0000	66
Virtualize	0.9571	1.0000	0.9781	67	VirtualizeAntiAliasAnalysis	1.0000	1.0000	1.0000	66

Table A1. *Cont.*

Class	Precision	Recall	F1-Score	Support	Class	Precision	Recall	F1-Score	Support
VirtualizeAntiTaintAnalysis	0.9833	0.8806	0.9291	67	VirtualizeEncodeArithmetic	1.0000	1.0000	1.0000	64
VirtualizeEncodeBranches	1.0000	0.9552	0.9771	67	VirtualizeEncodeLiterals	1.0000	1.0000	1.0000	66
VirtualizeFlatten	1.0000	1.0000	1.0000	67	VirtualizeSplit	1.0000	1.0000	1.0000	64
non-obfuscated	1.0000	0.9412	0.9697	17					

Table A2. Experiment 0 (CatBoost): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
CatBoost	0.9269	0.9282	0.9271	0.9273

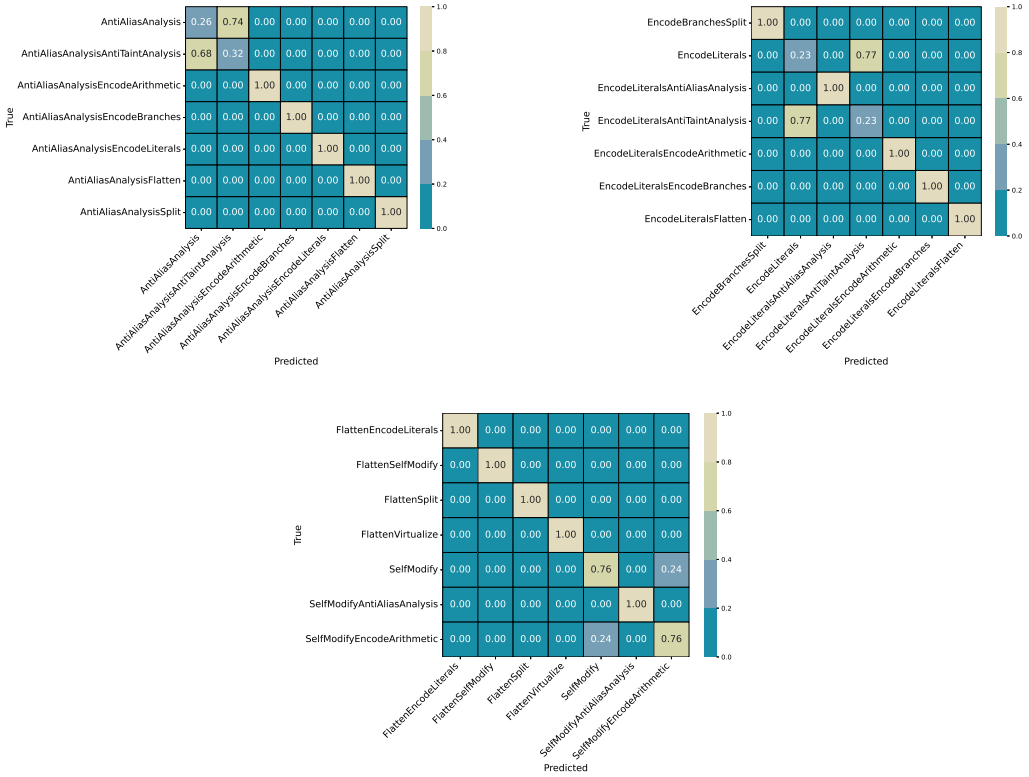

Figure A1. Experiment 0 (CatBoost): Confusion matrix excerpts (three snapshots from the full relative matrix). **Top row**: *AntiAliasAnalysis* vs. *AntiAliasAnalysisAntiTaintAnalysis* (**left**) and the *EncodeLiterals* cluster (**right**). **Bottom**: *SelfModify* cluster. Note that these confusion subsets do not fully represent the confusions across all classes. Some relative scores here are higher than in the whole picture, as some inter-class-confusions are not accounted for.

Appendix A.2. Cascading Experiments (CatBoost)

We repeat the staged inference workflow from the main text using CatBoost. Each model is trained on the same stratified split as in the ExtraTrees experiments and evaluated on the hold-out test set. The stages are identical:

- **Experiment 1:** Binary detection (*Obfuscated* vs. *Non_Obfuscated*).
- **Experiment 2:** Binary classification (*Single_Obf* vs. *Layered_Obf*).
- **Experiment 3:** Multiclass identification for *Single_Obf*.
- **Experiment 4:** Multiclass identification for *Layered_Obf*.

Performance metrics follow the same structure as in the ExtraTrees result (accuracy, precision, recall, and F1). All confusion matrices and detailed per-class reports are provided in the following . The final integrated pipeline results are summarized in Appendix A.3.

Appendix A.2.1. Experiment 1: Binary Obfuscation Detection (Obfuscated vs. Non-Obfuscated, CatBoost)

This binary classification has target `ObfBinary` ∈ {*Obfuscated*, *Non_Obfuscated*}. Per-class precision, recall, and F1 are listed in Table A3, summarized results in Table A4 and the corresponding confusion matrix is shown in Figure A2.

Table A3. Experiment 1 (CatBoost): Per-class report on the hold-out test split.

	Precision	Recall	F1-Score	Support
Non_Obfuscated	1.0000	0.8235	0.9032	17
Obfuscated	0.9993	1.0000	0.9997	4416

Table A4. Experiment 1 (CatBoost): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
CatBoost	0.9993	0.9997	0.9118	0.9514

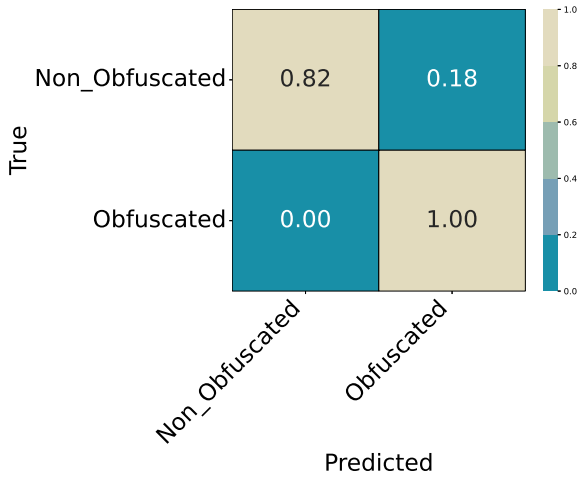

Figure A2. Experiment 1 (CatBoost): Confusion matrix for *Obfuscated* vs. *Non_Obfuscated* (relative).

Summary

CatBoost achieves an overall accuracy of **0.9993**. The *Obfuscated* class is detected with perfect recall, while the smaller *Non_Obfuscated* class achieves a recall of **0.8235**. This confirms that CatBoost, like ExtraTrees, cleanly separates obfuscated from non-obfuscated code, forming a reliable first stage in the cascading pipeline.

Appendix A.2.2. Experiment 2: Binary Single vs. Layered Obfuscations (CatBoost)

This experiment uses only obfuscated samples, targeting `SingleVsLayer` ∈ {*Single_Obf*, *Layered_Obf*}. Per-class results are listed in Table A5, summarized results in Table A6 and the confusion matrix is shown in Figure A3.

Table A5. Experiment 2 (CatBoost): Per-class report on the hold-out test split.

	Precision	Recall	F1-Score	Support
Layered_Obf	0.9520	0.9950	0.9731	3829
Single_Obf	0.9541	0.6729	0.7892	587

Table A6. Experiment 2 (CatBoost): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
CatBoost	0.9522	0.9531	0.8340	0.8811

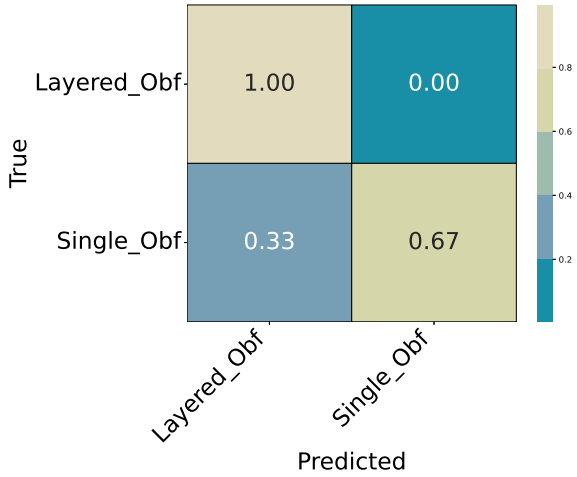

Figure A3. Experiment 2 (CatBoost): Confusion matrix for *Single_Obf* vs. *Layered_Obf* (relative).

Summary

CatBoost achieves an overall accuracy of **0.9522**. The model detects *Layered_Obf* samples with high recall (**0.9950**), while *Single_Obf* shows lower recall (**0.6729**). Most misclassifications occur when single obfuscations are mistaken for layered ones. This asymmetry is consistent with ExtraTrees and reflects the stronger signal of layered transformations compared to certain single methods, especially those with control-flow or literal encoding effects.

Appendix A.2.3. Experiment 3: Single-Obfuscation Identification (Multiclass, CatBoost)

This experiment isolates all samples labeled as *Single_Obf* and identifies the specific transformation applied. The target variable `SingleMethod` covers nine canonical obfuscation techniques. Per-class metrics are reported in Tables A7 and A8, and the confusion matrix is shown in Figure A4.

Table A7. Experiment 3 (CatBoost): Per-class report on the hold-out test split.

	Precision	Recall	F1-Score	Support
AntiAliasAnalysis	1.0000	1.0000	1.0000	62
AntiTaintAnalysis	1.0000	1.0000	1.0000	67
EncodeArithmetic	1.0000	1.0000	1.0000	64
EncodeBranches	1.0000	1.0000	1.0000	67
EncodeLiterals	1.0000	1.0000	1.0000	66
Flatten	1.0000	1.0000	1.0000	67
SelfModify	1.0000	1.0000	1.0000	63
Split	1.0000	1.0000	1.0000	64
Virtualize	1.0000	1.0000	1.0000	67

Table A8. Experiment 3 (CatBoost): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
CatBoost	1.0000	1.0000	1.0000	1.0000

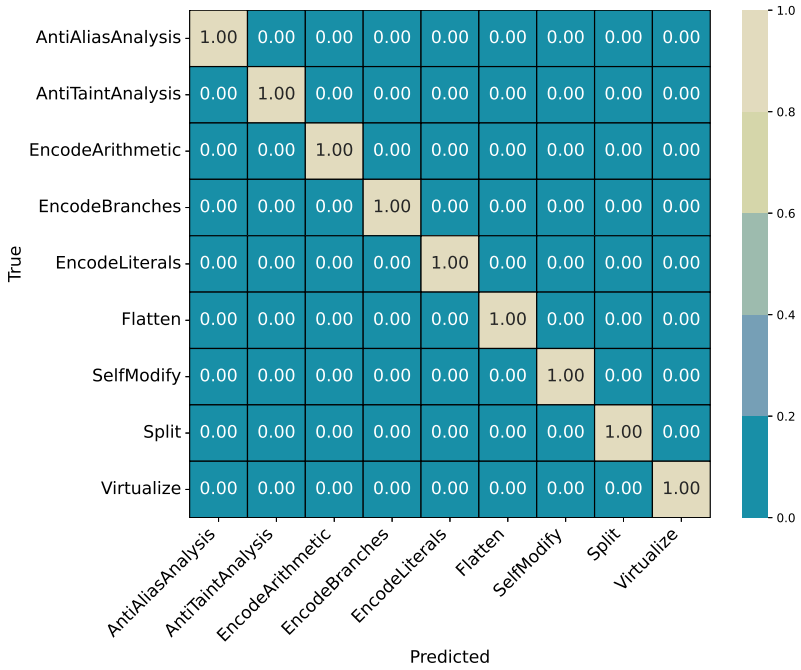

Figure A4. Experiment 3 (CatBoost): Confusion matrix over all single obfuscation classes (relative).

Summary

CatBoost achieves perfect accuracy (**1.0000**) across all nine single obfuscation types, matching the performance of ExtraTrees. All classes show full precision, recall, and F1, confirming complete separability in the IR2Vec feature space. The confusion matrix (Figure A4) exhibits a clean diagonal structure, indicating that each transformation produces distinct feature patterns.

Appendix A.2.4. Experiment 4: Layered-Obfuscation Identification (Multiclass, CatBoost)

This experiment uses only samples labeled as *Layered_Obf* and identifies the specific combination of transformations applied. The target variable `LayeredLabel` encodes each unique layered configuration. Per-class precision, recall, and F1-scores are shown in Table A9, while sumamry results are given in Table A10.

Summary

CatBoost achieves an overall accuracy of **0.9851**, showing stable and near-uniform results across all layered configurations. Most combinations are identified without error, while minor confusion occurs between closely related variants such as *FlattenAntiTaintAnalysis* and *EncodeArithmeticAntiTaintAnalysis*.

Table A9. Experiment 4 (CatBoost): Per-class report on the hold-out test split.

Class	Precision	Recall	F1-Score	Support	Class	Precision	Recall	F1-Score	Support
AntiAliasAnalysisAntiTaintAnalysis	1.0000	1.0000	1.0000	66	EncodeBranchesEncodeArithmetic	0.9833	0.9219	0.9516	64
AntiAliasAnalysisEncodeArithmetic	1.0000	1.0000	1.0000	62	EncodeBranchesEncodeLiterals	1.0000	1.0000	1.0000	66
AntiAliasAnalysisEncodeBranches	1.0000	1.0000	1.0000	66	EncodeBranchesSplit	0.9692	0.9844	0.9767	64
AntiAliasAnalysisEncodeLiterals	1.0000	1.0000	1.0000	66	EncodeLiteralsAntiAliasAnalysis	1.0000	1.0000	1.0000	62
AntiAliasAnalysisFlatten	1.0000	1.0000	1.0000	66	EncodeLiteralsAntiTaintAnalysis	1.0000	1.0000	1.0000	66
AntiAliasAnalysisSplit	1.0000	1.0000	1.0000	62	EncodeLiteralsEncodeArithmetic	1.0000	1.0000	1.0000	62
AntiAliasAnalysisVirtualize	1.0000	1.0000	1.0000	66	EncodeLiteralsEncodeBranches	1.0000	1.0000	1.0000	66
AntiTaintAnalysisEncodeArithmetic	0.9265	0.9844	0.9545	64	EncodeLiteralsFlatten	1.0000	1.0000	1.0000	66
AntiTaintAnalysisEncodeBranches	1.0000	0.9851	0.9925	67	EncodeLiteralsSplit	1.0000	1.0000	1.0000	62
AntiTaintAnalysisFlatten	0.7945	0.8657	0.8286	67	EncodeLiteralsVirtualize	1.0000	1.0000	1.0000	66
AntiTaintAnalysisSplit	0.9014	1.0000	0.9481	64	FlattenAntiAliasAnalysis	1.0000	1.0000	1.0000	66
AntiTaintAnalysisVirtualize	0.9167	0.9851	0.9496	67	FlattenAntiTaintAnalysis	0.8525	0.7761	0.8125	67

Table A9. *Cont.*

Class	Precision	Recall	F1-Score	Support	Class	Precision	Recall	F1-Score	Support
EncodeArithmeticAntiAliasAnalysis	1.0000	1.0000	1.0000	62	FlattenEncodeArithmetic	1.0000	0.9851	0.9925	67
EncodeArithmeticAntiTaintAnalysis	0.9841	0.9254	0.9538	67	FlattenEncodeBranches	1.0000	1.0000	1.0000	67
EncodeArithmeticEncodeBranches	0.9296	0.9851	0.9565	67	FlattenEncodeLiterals	1.0000	1.0000	1.0000	66
EncodeArithmeticEncodeLiterals	1.0000	1.0000	1.0000	66	FlattenSelfModify	1.0000	1.0000	1.0000	66
EncodeArithmeticFlatten	0.9853	1.0000	0.9926	67	FlattenSplit	1.0000	1.0000	1.0000	64
EncodeArithmeticSelfModify	1.0000	1.0000	1.0000	62	FlattenVirtualize	1.0000	1.0000	1.0000	67
EncodeArithmeticSplit	0.9697	1.0000	0.9846	64	SelfModifyAntiAliasAnalysis	1.0000	1.0000	1.0000	62
EncodeArithmeticVirtualize	1.0000	1.0000	1.0000	67	SelfModifyEncodeArithmetic	1.0000	1.0000	1.0000	63
EncodeBranchesAntiAliasAnalysis	1.0000	1.0000	1.0000	62	SplitAntiAliasAnalysis	1.0000	1.0000	1.0000	62
EncodeBranchesAntiTaintAnalysis	0.9853	1.0000	0.9926	67	SplitAntiTaintAnalysis	1.0000	0.8906	0.9421	64
TigressRecipe1	1.0000	1.0000	1.0000	62	SplitEncodeArithmetic	1.0000	0.9688	0.9841	64
TigressRecipe2	1.0000	1.0000	1.0000	66	SplitEncodeBranches	0.9841	0.9688	0.9764	64
VirtualizeAntiAliasAnalysis	1.0000	1.0000	1.0000	66	VirtualizeAntiTaintAnalysis	0.9839	0.9104	0.9457	67
VirtualizeEncodeArithmetic	1.0000	1.0000	1.0000	64	VirtualizeEncodeBranches	1.0000	1.0000	1.0000	67
VirtualizeEncodeLiterals	1.0000	1.0000	1.0000	66	VirtualizeFlatten	1.0000	1.0000	1.0000	67
VirtualizeSplit	1.0000	1.0000	1.0000	64					

Table A10. Experiment 4 (CatBoost): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
CatBoost	0.9851	0.9859	0.9854	0.9853

Appendix A.3. Final Cascaded Inference (Integrated Pipeline, CatBoost)

We evaluate the full CatBoost cascade on the same hold-out split. The pipeline applies stage models in sequence: obfuscation detection, single vs. layered (for obfuscated samples), and method identification using the corresponding single/layered model. Table A11 lists the per-class report in a compact two-panel layout; summaries are in Table A12. Figure A5 shows three confusion-matrix excerpts.

Table A11. Final cascading classification report (CatBoost).

Class	Precision	Recall	F1-Score	Support	Class	Precision	Recall	F1-Score	Support
AntiAliasAnalysis	0.0000	0.0000	0.0000	62	AntiAliasAnalysisAntiTaintAnalysis	0.4672	0.8636	0.6064	66
AntiAliasAnalysisEncodeArithmetic	1.0000	1.0000	1.0000	62	AntiAliasAnalysisEncodeBranches	0.8571	0.9091	0.8824	66
AntiAliasAnalysisEncodeLiterals	1.0000	1.0000	1.0000	66	AntiAliasAnalysisFlatten	1.0000	1.0000	1.0000	66
AntiAliasAnalysisSplit	1.0000	1.0000	1.0000	62	AntiAliasAnalysisVirtualize	1.0000	1.0000	1.0000	66
AntiTaintAnalysis	1.0000	0.7910	0.8833	67	AntiTaintAnalysisEncodeArithmetic	0.8873	0.9844	0.9333	64
AntiTaintAnalysisEncodeBranches	0.8400	0.9403	0.8873	67	AntiTaintAnalysisFlatten	0.7941	0.8060	0.8000	67
AntiTaintAnalysisSplit	0.8000	1.0000	0.8889	64	AntiTaintAnalysisVirtualize	0.8649	0.9552	0.9078	67
EncodeArithmetic	0.9688	0.9688	0.9688	64	EncodeArithmeticAntiAliasAnalysis	1.0000	1.0000	1.0000	62
EncodeArithmeticAntiTaintAnalysis	0.9833	0.8806	0.9291	67	EncodeArithmeticEncodeBranches	0.9041	0.9851	0.9429	67
EncodeArithmeticEncodeLiterals	1.0000	1.0000	1.0000	66	EncodeArithmeticFlatten	0.9853	1.0000	0.9926	67
EncodeArithmeticSelfModify	0.9841	1.0000	0.9920	62	EncodeArithmeticSplit	0.9697	1.0000	0.9846	64
EncodeArithmeticVirtualize	0.9710	1.0000	0.9853	67	EncodeBranches	0.9853	1.0000	0.9926	67
EncodeBranchesAntiAliasAnalysis	1.0000	1.0000	1.0000	62	EncodeBranchesAntiTaintAnalysis	0.8767	0.9552	0.9143	67
EncodeBranchesEncodeArithmetic	0.9833	0.9219	0.9516	64	EncodeBranchesEncodeLiterals	1.0000	1.0000	1.0000	66
EncodeBranchesSplit	0.9048	0.8906	0.8976	64	EncodeLiterals	0.0000	0.0000	0.0000	66
EncodeLiteralsAntiAliasAnalysis	1.0000	1.0000	1.0000	62	EncodeLiteralsAntiTaintAnalysis	0.4848	0.9697	0.6465	66
EncodeLiteralsEncodeArithmetic	1.0000	1.0000	1.0000	62	EncodeLiteralsEncodeBranches	0.9697	0.9697	0.9697	66
EncodeLiteralsFlatten	1.0000	1.0000	1.0000	66	EncodeLiteralsSplit	1.0000	1.0000	1.0000	62
EncodeLiteralsVirtualize	1.0000	1.0000	1.0000	66	Flatten	1.0000	1.0000	1.0000	67
FlattenAntiAliasAnalysis	1.0000	1.0000	1.0000	66	FlattenAntiTaintAnalysis	0.8030	0.7910	0.7970	67
FlattenEncodeArithmetic	1.0000	0.9851	0.9925	67	FlattenEncodeBranches	1.0000	1.0000	1.0000	67
FlattenEncodeLiterals	1.0000	1.0000	1.0000	66	FlattenSelfModify	1.0000	1.0000	1.0000	66
FlattenSplit	1.0000	1.0000	1.0000	64	FlattenVirtualize	1.0000	1.0000	1.0000	67
SelfModify	0.8333	0.7143	0.7692	63	SelfModifyAntiAliasAnalysis	1.0000	1.0000	1.0000	62
SelfModifyEncodeArithmetic	0.7606	0.8571	0.8060	63	Split	0.8431	0.6719	0.7478	64
SplitAntiAliasAnalysis	1.0000	1.0000	1.0000	62	SplitAntiTaintAnalysis	1.0000	0.7500	0.8571	64
SplitEncodeArithmetic	0.7826	0.8438	0.8120	64	SplitEncodeBranches	0.8732	0.9688	0.9185	64
SplitEncodeLiterals	1.0000	1.0000	1.0000	62	SplitFlatten	1.0000	1.0000	1.0000	64
SplitSelfModify	1.0000	1.0000	1.0000	63	SplitVirtualize	1.0000	1.0000	1.0000	64
TigressRecipe1	1.0000	1.0000	1.0000	62	TigressRecipe2	1.0000	1.0000	1.0000	66
Virtualize	1.0000	0.8657	0.9280	67	VirtualizeAntiAliasAnalysis	1.0000	1.0000	1.0000	66
VirtualizeAntiTaintAnalysis	0.9500	0.8507	0.8976	67	VirtualizeEncodeArithmetic	1.0000	1.0000	1.0000	64
VirtualizeEncodeBranches	0.9054	1.0000	0.9504	67	VirtualizeEncodeLiterals	1.0000	1.0000	1.0000	66
VirtualizeFlatten	1.0000	1.0000	1.0000	67	VirtualizeSplit	1.0000	1.0000	1.0000	64
non-obfuscated	1.0000	0.8235	0.9032	17					

Table A12. Final cascade summary (CatBoost).

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
CatBoost	0.9276	0.9164	0.9263	0.9179

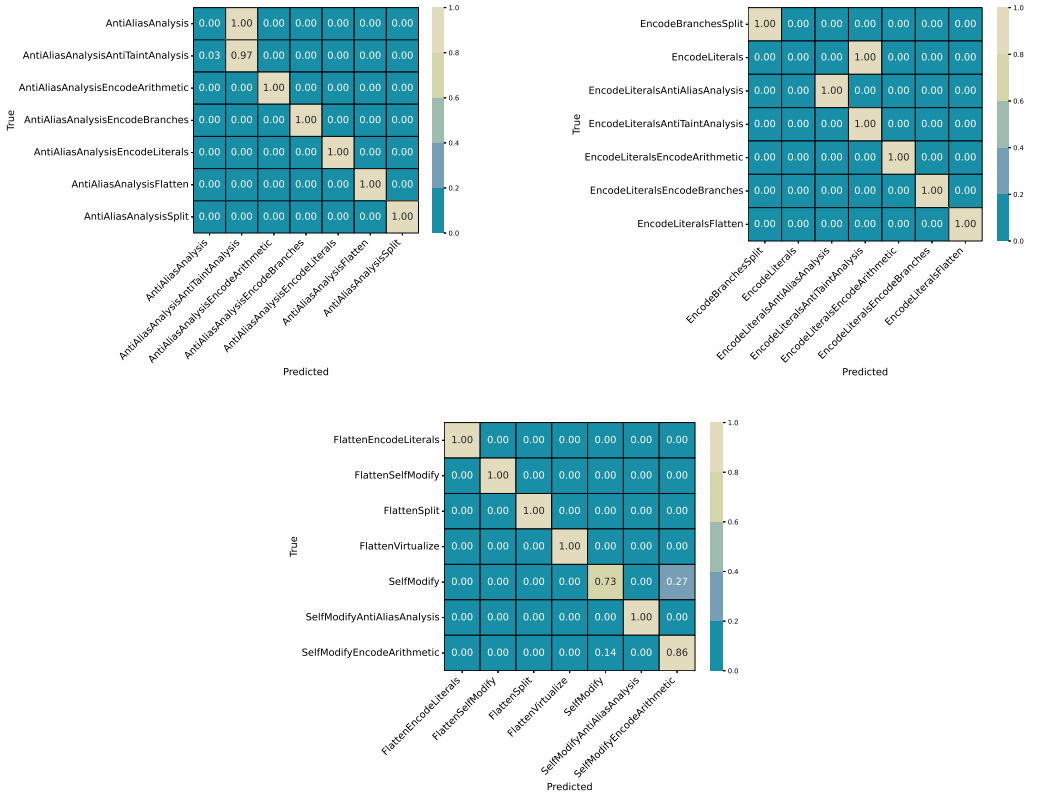

Figure A5. Final cascade (CatBoost): Confusion-matrix excerpts (relative). **Top**: *AntiAliasAnalysis* vs. *AntiAliasAnalysisAntiTaintAnalysis* (**left**) and *EncodeLiterals* cluster (**right**). **Bottom**: *SelfModify* cluster. Note that these confusion subsets do not fully represent the confusions across all classes. Some relative scores here are higher than in the whole picture, as some inter-class-confusions are not accounted for.

Summary

The CatBoost cascade attains **92.76%** accuracy (macro F1 **0.9179**). Most layered and many single-pass classes remain strong. Weak classes mirror earlier trends: *AntiAliasAnalysis* and base *EncodeLiterals* show near-zero recall, and *Virtualize* and *AntiTaintAnalysis* have reduced recall compared to ExtraTrees. *Non_Obfuscated* recall is 0.8235 on its small support. Overall, the end-to-end performance is consistent with the CatBoost stage results and lower than ExtraTrees, but the pipeline composes reliably.

Appendix B. O-Level Identification (Experiment 5)

This appendix presents the results for Experiment 5, where the goal is to identify compiler optimization levels (*O-levels*) from LLVM IR samples. The experiment is designed to test whether optimization levels (`O1`, `O2`, `O3`) can be reliably detected and distinguished from non-optimized (`No_O`) samples. The task is treated as a multiclass classification problem using the same train/test split and preprocessing pipeline as all previous experiments.

We report results separately for the ExtraTrees and CatBoost classifiers. Each subsection provides a preliminary classification report with placeholder performance values and a confusion matrix visualization.

Appendix B.1. ExtraTrees Classifier (O-Level Identification)

This experiment evaluates the capability of the ExtraTrees classifier to identify compiler optimization levels (`O0`–`O3`) and distinguish them from samples compiled without explicit optimization (`No_O`). The classification report is summarized in Table A13, while overall metrics are presented in Table A14. The corresponding confusion matrix for the best-performing ExtraTrees model is shown in Figure A6.

Table A13. Experiment 5 (ExtraTrees): Per-class report on the hold-out test split.

	Precision	Recall	F1-Score	Support
No_O	1.0000	0.1765	0.3000	17
O0	0.2135	0.3803	0.2735	1099
O1	0.0405	0.0027	0.0051	1106
O2	0.0448	0.0026	0.0049	1147
O3	0.2051	0.4492	0.2816	1064

Table A14. Experiment 5 (ExtraTrees): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
ExtraTrees	0.2042	0.3008	0.2023	0.1730

Summary

The classifier achieves an overall accuracy of only **20.42%**, indicating that identifying compiler optimization levels is substantially more challenging than detecting obfuscation types. Although the model partially separates the extremes (*No_O* and *O3*) confusions between intermediate levels (*O0*–*O2*) dominate (Figure A6). This pattern suggests that, in contrast to obfuscation detection, optimization-level identification lacks distinct IR-level cues and is therefore not reliably solvable with the same representation and feature set.

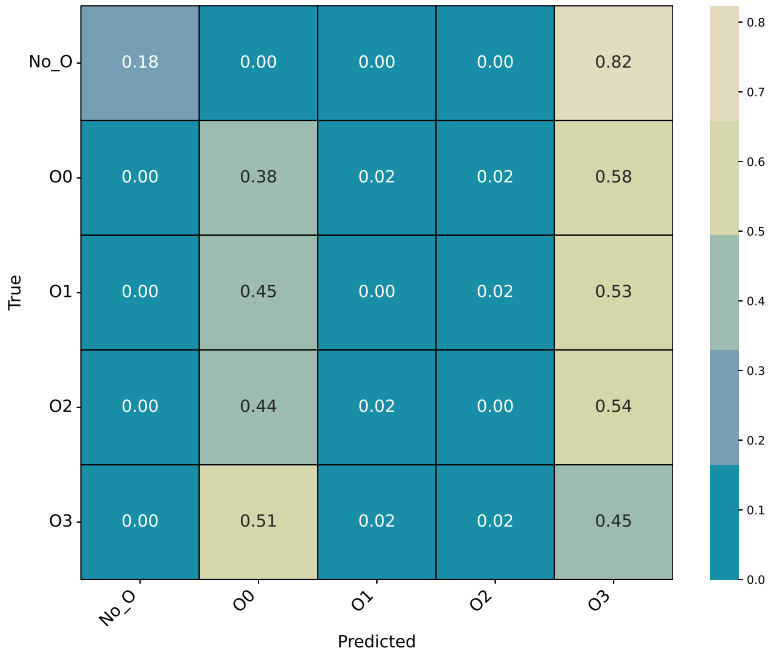

Figure A6. Experiment 5 (ExtraTrees): Confusion matrix for O-level identification (*No_O*, *O0*, *O1*, *O2*, *O3*).

Appendix B.2. CatBoost Classifier (O-Level Identification)

The CatBoost model was trained under identical conditions as the ExtraTrees baseline and evaluated on the same hold-out test split. This experiment again targets compiler optimization levels (`O0`–`O3`) and the non-optimized class (`No_O`). The detailed per-class metrics are listed in Table A15, overall performance metrics are shown in Table A16, and the confusion matrix is presented in Figure A7.

Table A15. Experiment 5 (CatBoost): Per-class report on the hold-out test split.

	Precision	Recall	F1-Score	Support
No_O	1.0000	0.8824	0.9375	17
O0	0.0236	0.0255	0.0245	1099
O1	0.0170	0.0163	0.0167	1106
O2	0.0178	0.0157	0.0167	1147
O3	0.0188	0.0207	0.0197	1064

Table A16. Experiment 5 (CatBoost): Summary metrics on the hold-out test split.

	Accuracy	Precision (Macro)	Recall (Macro)	F1 (Macro)
CatBoost	0.0228	0.2155	0.1921	0.2030

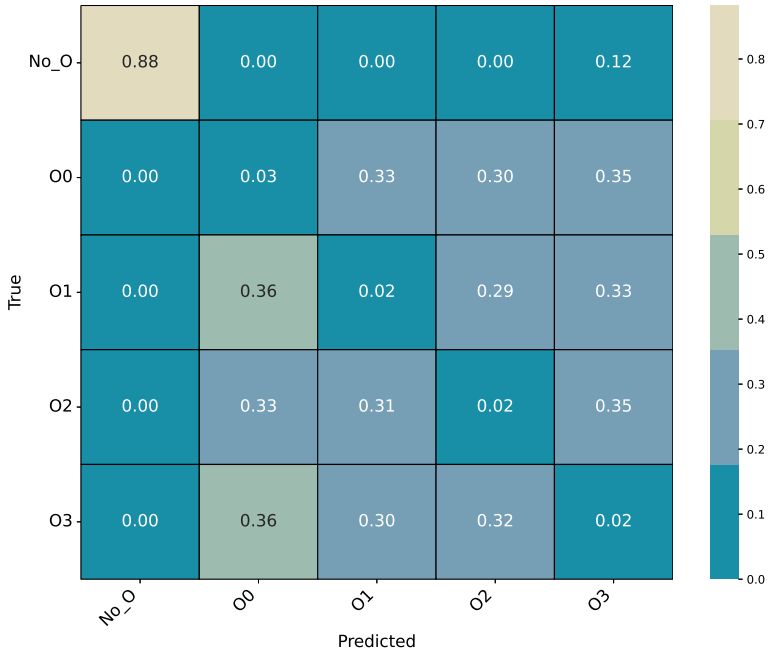

Figure A7. Experiment 5 (CatBoost): Confusion matrix for O-level identification (*No_O*, *O0*, *O1*, *O2*, *O3*).

Summary

The CatBoost classifier performs significantly worse than the ExtraTrees model for O-level identification, reaching an overall accuracy of only **2.28%**. Although *No_O* samples are detected with relatively high recall (0.88), all optimization-level classes (*O0–O3*) are essentially indistinguishable, with near-random classification performance. This outcome further confirms that the IR-level representation contains insufficient structural variation to capture compiler optimization differences. Consequently, both CatBoost and ExtraTrees fail to generalize meaningfully for O-level prediction, supporting the conclusion that this task cannot be reliably solved in the current feature space.

References

1. 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, Orlando, FL, USA, 3–7 December 2012; pp. 319–328.
2. Junod, P.; Rinaldini, J.; Wehrli, J.; Michielin, J. Obfuscator-LLVM—Software protection for the masses. In Proceedings of the 2015 IEEE/ACM 1st International Workshop on Software Protection, Florence, Italy, 19 May 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 3–9.
3. 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, Los Angeles, CA, USA, 5–6 December 2016; pp. 1–11.
4. Sagisaka, H.; Tamada, H. Identifying the applied obfuscation method towards de-obfuscation. In Proceedings of the 2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS), Okayama, Japan, 26–29 June 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1–6.
5. Tesauro, G.J.; Kephart, J.O.; Sorkin, G.B. Neural networks for computer virus recognition. *IEEE Expert* **1996**, *11*, 5–6. [CrossRef]
6. Jones, L.; Christman, D.; Banescu, S.; Carlisle, M. Bytewise: A case study in neural network obfuscation identification. In Proceedings of the 2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 8–10 January 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 155–164.
7. Kim, J.; Kang, S.; Cho, E.S.; Paik, J.Y. LOM: Lightweight Classifier for Obfuscation Methods. In *Information Security Applications, Proceedings of the 22nd International Conference, WISA 2021, Jeju Island, Republic of Korea, 11–13 August 2021*; Kim, H., Ed.; Springer: Cham, Switzerland, 2021; pp. 3–15.
8. Schrittwieser, S.; Wimmer, E.; Mallinger, K.; Kochberger, P.; Lawitschka, C.; Raubitzek, S.; Weippl, E.R. Modeling Obfuscation Stealth Through Code Complexity. In Proceedings of the Computer Security. ESORICS 2023 International Workshops: CPS4CIP, ADIoT, SecAssure, WASP, TAURIN, PriST-AI, and SECAI, The Hague, The Netherlands, 25–29 September 2023; Revised Selected Papers, Part II; Springer: Berlin/Heidelberg, Germany, 2023; pp. 392–408. [CrossRef]
9. VenkataKeerthy, S.; Aggarwal, R.; Jain, S.; Desarkar, M.S.; Upadrasta, R.; Srikant, Y.N. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* **2020**, *17*, 1–27. [CrossRef]
10. Raubitzek, S.; Schrittwieser, S.; Wimmer, E.; Mallinger, K. Obfuscation undercover: Unraveling the impact of obfuscation layering on structural code patterns. *J. Inf. Secur. Appl.* **2024**, *85*, 103850. [CrossRef]
11. Raubitzek, S.; Schrittwieser, S.; Lawitschka, C.; Mallinger, K.; Ekelhart, A.; Weippl, E.R. Code Obfuscation Classification Using Singular Value Decomposition on Grayscale Image Representations. In Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT), Dijon, France, 8–10 July 2024; p. TBD.
12. Raitsis, T.; Elgazari, Y.; Toibin, G.E.; Lurie, Y.; Mark, S.; Margalit, O. Code Obfuscation: A Comprehensive Approach to Detection, Classification, and Ethical Challenges. *Algorithms* **2025**, *18*, 54. [CrossRef]
13. Jiang, S.; Hong, Y.; Fu, C.; Qian, Y.; Han, L. Function-level obfuscation detection method based on Graph Convolutional Networks. *J. Inf. Secur. Appl.* **2021**, *61*, 102953. [CrossRef]
14. Wang, Y.; Rountev, A. Who changed you? Obfuscator identification for Android. In Proceedings of the 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Buenos Aires, Argentina, 22–23 May 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 154–164.
15. 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, Hamburg Germany, 27–30 August 2018; pp. 1–9.
16. Park, M.; You, G.; Cho, S.j.; 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*, 22–30.
17. Collberg, C. Tigress: The Diversifying C Virtualizer/Obfuscator. 2025. Available online: https://tigress.wtf/ (accessed on 10 October 2025).
18. Banescu, S.; Ochoa, M.; Pretschner, A. A framework for measuring software obfuscation resilience against automated attacks. In Proceedings of the 2015 IEEE/ACM 1st International Workshop on Software Protection, Florence, Italy, 19 May 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 45–51.
19. Patra, B.; Kanade, A.; Maniatis, P.; Orso, A. IR2Vec: A Flow-Aware Representation Learning for LLVM IR. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020), Virtual Event, 8–13 November 2020; ACM: Singapore, 2020; pp. 234–245. [CrossRef]
20. Banescu, S.; Collberg, C.; Pretschner, A. Measuring the Strength of Software Obfuscation. In Proceedings of the 2016 Workshop on Software Protection (SPRO '16), Vienna, Austria, 28 October 2016; pp. 1–12.
21. Salem, M.; Pretschner, A.; Banescu, S. Machine Learning-Based Detection of Software Obfuscation Techniques. *J. Inf. Secur. Appl.* **2019**, *46*, 76–89.

22. Prokhorenkova, L.; Gusev, G.; Vorobev, A.; Dorogush, A.V.; Gulin, A. CatBoost: Unbiased boosting with categorical features. In Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), Montreal, QC, Canada, 3–8 December 2018; pp. 6638–6648.
23. Dorogush, A.V.; Ershov, V.; Gulin, A. CatBoost: Gradient boosting with categorical features support. In Proceedings of the Workshop on ML Systems at NeurIPS, 2017, Long Beach Convention Center, Long Beach, California, USA, December 4–9, 2017 .
24. Geurts, P.; Ernst, D.; Wehenkel, L. Extremely Randomized Trees. *Mach. Learn.* **2006**, *63*, 3–42. [CrossRef]
25. Raubitzek, S.; Corpaci, L.; Hofer, R.; Mallinger, K. Scaling Exponents of Time Series Data: A Machine Learning Approach. *Entropy* **2023**, *25*, 1671. [CrossRef] [PubMed]
26. Raubitzek, S.; Mallinger, K. On the Applicability of Quantum Machine Learning. *Entropy* **2023**, *25*, 992. [CrossRef] [PubMed]
27. Snoek, J.; Larochelle, H.; Adams, R.P. Practical Bayesian Optimization of Machine Learning Algorithms. *arXiv* **2012**, arXiv:1206.2944. [CrossRef]
28. Head, T.; Kumar, M.; Nahrstaedt, H.; Louppe, G.; Shcherbatyi, I. Scikit-Optimize/Scikit-Optimize (v0.9.0). 2021. Available online: https://zenodo.org/records/5565057 (accessed on 10 October 2025).