SERBIAN JOURNAL OF ELECTRICAL ENGINEERING
Vol. 22, No. 3, October 2025, 353-376

UDC: 004.85: 004.4`422 **DOI: https://doi.org/10.2298/SJEE250828003R**

Original scientific paper

Machine Learning-Driven Prediction of Optimal Control Flow Graph Traversal Strategy

Ivan Ristović[1,2], Milan Čugurović[1,2], Strahinja Stanojević[1,2], Marko Spasić[1,2], Vesna Marinković[1], Milena Vujošević Janičić[1,2]

Abstract: Control flow graphs model possible program execution paths and thus are essential for static program analysis. Compilers use control flow graphs as a basis for their intermediate representations, allowing them to apply optimizations. As each method is represented by its control flow graph, the number of control flow graphs that a compiler needs to generate and process depends on the program being compiled. For reference, modern programs that run on the JVM consist of hundreds of thousands of methods. Thus, efficient control flow graph traversal is crucial to provide fast compilation. Prior work has shown that breadth-first and depth-first search algorithms yield different results depending on the control flow graph structure; however, the relationship between control flow graph features and the optimal traversal algorithm in terms of traversal speed remains underexplored. In this work, we construct a dataset of over 200,000 control flow graphs gathered from modern state-of-the-art JVM benchmark suites. Using this dataset, we train a set of ensemble-based machine learning models that predict optimal graph traversal algorithms for a given control flow graph using a set of lightweight graph features. Our models identify the key features that yield accurate predictions and demonstrate that the most informative features can be extracted efficiently during the graph construction process itself.

Keywords: Compilers, Machine Learning, Control Flow Graphs, Graph Traversals, GraalVM.

[1]Faculty of Mathematics, University of Belgrade, Serbia
milan.cugurovic@matf.bg.ac.rs, https://orcid.org/0009-0003-4149-5820
ivan.ristovic@matf.bg.ac.rs, https://orcid.org/0000-0002-1679-3848
strahinja.stanojevic@matf.bg.ac.rs, https://orcid.org/0009-0007-6076-3586
marko.spasic@matf.bg.ac.rs, https://orcid.org/0009-0000-0392-0935
vesna.marinkovic@matf.bg.ac.rs, https://orcid.org/0000-0003-0526-899X
milena.vujosevic.janicic@matf.bg.ac.rs, https://orcid.org/0000-0001-5396-0644
[2]Oracle Labs, Belgrade, Serbia

*An earlier version of this paper was presented at the 12th International Conference on Electrical, Electronic and Computing Engineering (IcETRAN), Čačak, Serbia, June 2025.

Colour versions of the one or more of the figures in this paper are available online at https://sjee.ftn.kg.ac.rs

1 Introduction

Graphs are a fundamental abstraction in computer science, commonly used to represent data and the relationships between data [1]. Among their broad scope of applications, control flow graphs (CFGs) are particularly significant in semantic analysis during compilation, or malware detection [2 – 4]. CFG models the different paths a program might take during its execution, serving as a foundation for various analysis and optimization techniques [5]. In compiler ecosystems like GraalVM [6, 7], CFGs are crucial in guiding code transformations and performance improvements [8]. Given the complexity of modern applications, compilers frequently process hundreds of thousands of control flow graphs throughout the compilation process [9, 10].

Efficient traversal of control flow graphs is a key requirement for optimizing compilation performance [11]. Prior research has explored how the structural properties of CFGs influence the efficiency of different traversal strategies [12 – 14]. Additionally, spectral graph theory has provided valuable mathematical tools for capturing and analyzing the complexity of graph structures, including CFGs [15, 16]. Recent approaches also leverage machine learning techniques to classify CFGs and extract meaningful structural and semantic features [17]. While performance trade-offs between traversal algorithms such as depth-first search (DFS) and breadth-first search (BFS) have been studied in various contexts [18], to the best of our knowledge, no prior work has specifically compared these algorithms in the context of CFG traversal for Java and Scala programs. Although both DFS and BFS operate in linear time in theory, their actual performance can differ significantly based on the topology of the CFG being traversed [19].

Extracting certain properties of control flow graphs often necessitates traversing the graph itself [13]. This presents a fundamental paradox: identifying whether DFS or BFS is more efficient for a given control flow graph typically requires performing a traversal in the first place. At first glance, this seems counterintuitive — if traversal is needed to make the decision, any benefit from choosing the optimal strategy appears negated. However, our work challenges this assumption by proposing a predictive approach. Instead of analyzing each new graph from scratch, we explore whether it is possible to infer the optimal traversal strategy based on ML predictions. The predictions are based on CFG features that can be obtained without requiring additional graph traversals. This transforms what initially appears to be a redundant process into a practical optimization tool. This approach is especially valuable in performance-critical environments, like compilers and static analysis tools, where even small efficiency improvements can accumulate into significant overall speed-ups when applied across many CFGs.

In this work, we expand our previous findings [14] that introduce an ensemble-based machine learning model designed to predict the most efficient traversal algorithm, depth-first or breadth-first, for a given control flow graph. We describe an extensible pipeline used to extract CFG properties and modify our dataset to make it more generic. We provide descriptions of measurement techniques that can be used to extract desired target metrics. With these techniques, we compiled a labeled dataset containing over 220,000 CFGs derived from modern Java and Scala applications [9, 10], used for model training. By leveraging machine learning, we not only automate the selection of the optimal traversal strategy but also gain valuable insights into which CFG characteristics are most influential for this decision. Importantly, we demonstrate that many of these features can be gathered during the graph construction phase itself, avoiding any redundant overhead. This enables an efficient ahead-of-time prediction for static graphs and supports on-the-fly feature extraction for newly generated or modified CFGs, making the approach practical for integration into compilers or static analysis tools.

The main contributions of this paper are:

– An extensible pipeline for extracting CFG features and dataset creation.
– A set of lightweight features that characterize CFGs and enable accurate prediction of graph traversal strategies.
– A labeled dataset of over 200,000 CFGs extracted from modern JVM-based applications [20], supporting various types of CFG analyses.
– A set of ensemble-based machine learning models that predict the optimal CFG traversal algorithm based solely on structural graph features.
– ML-driven insights into CFGs, identifying the number of nodes, average in-degree, and the number of non-binary splits as key features for the prediction of the optimal graph traversal algorithm.

2 Background

CFG analysis [5, 8] enables compilers, static profilers, and a wide range of program analysis tools to gain deep insights into the structure and control logic of a program without requiring its execution. This form of analysis and classification is commonly applied in contexts such as data flow analysis, symbolic execution, formal program verification, malware detection, and software vulnerability identification [21]. In practice, analyzing a CFG typically involves either extracting informative structural features or applying graph transformations, both of which inherently require traversing the CFG.

2.1 Control flow graphs and graph traversal algorithms

In a control flow graph, each node represents a *basic block* — a straight-line sequence of instructions with a single entry point and a single exit, containing no internal jumps or branches. The edges between these nodes represent the possible paths of control flow from one basic block to another.

Within compiler infrastructures, CFGs serve as a fundamental tool for program analysis and optimization, enabling a range of transformations and performance improvements. During the compilation process, each method in a program is typically converted into its own CFG. The graph starts with a designated entry node, corresponding to the beginning of the method, and includes one or more exit nodes that capture all possible termination points. CFGs can contain cycles when the method contains loops or recursive calls, or be acyclic when there are no such control structures.

Fig. 1 shows the source code of the *java.util.DualPivotQuicksort.sort* method from the Java standard library, while Fig. 2 represents the corresponding high-level CFG. The sort method conditionally invokes either the *countingSort* or *insertionSort* algorithm, depending on the number of elements to be sorted, as determined by the *if* statement. In the CFG, the conditional node serves as the common predecessor and has two successor nodes, one invoking *countingSort*, the other *insertionSort*, which then merge at a single exit node representing the method's end.

```
static void sort(byte[] a, int low, int high) {
    if (high - low > MIN_BYTE_COUNTING_SORT_SIZE) {
        countingSort(a, low, high);
    } else {
        insertionSort(a, low, high);
    }
}
```

Fig. 1 – *Source code of a standard Java library method java.util.DualPivotQuicksort.sort.*

Compiler optimizations often rely on traversing the control flow graph to gather relevant information or apply various code transformations. Traversal, in this context, means visiting all nodes that are reachable from the graph's entry (start) node. The two most widely used traversal algorithms are DFS and BFS [1], both of which operate in linear time with respect to the number of nodes and edges in the graph. Despite having identical asymptotic complexity, their practical performance can vary significantly depending on the structure and characteristics of the CFG, with one traversal method often proving more efficient than the other in certain scenarios.

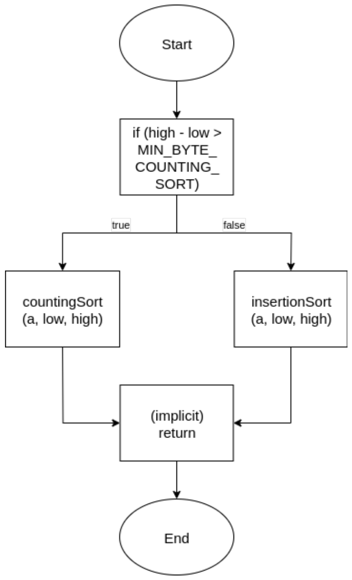
Start
if (high - low > MIN_BYTE_ COUNTING_ SORT)
true
false
countingSort (a, low, high)
insertionSort (a, low, high)
(implicit) return
End

Fig. 2 – *An example of a high-level Control Flow Graph corresponding to Fig. 1.*

2.2 GraalVM compiler infrastructure

GraalVM [6, 7] is a powerful and versatile compiler infrastructure that combines several advanced components built around Graal [22], a high-performance, optimizing compiler written in Java. GraalVM supports both Just-In-Time (JIT) and Ahead-of-Time (AOT) compilation modes, leveraging the Graal compiler. The GraalVM Native Image tool [7] performs AOT compilation combined with class initialization at build time [23] to generate compact, standalone, and platform-specific executables called native images. Native images offer reduced startup time and lower memory overhead compared to traditional deployments [24], making them especially suitable for resource-constrained or latency-sensitive environments such as cloud computing [25].

At the heart of Graal's optimization pipeline lies its sea-of-nodes intermediate representation (IR) [26], known as GraalIR [22]. This IR merges control flow and data flow into a single, unified graph-based structure, enabling more powerful and flexible optimizations. In this representation, control flow nodes correspond to fixed execution points (such as basic block boundaries), while data flow nodes are floating, meaning they are not tied to a specific execution order and represent computed values, conditions, and branching logic. During the compilation process, Graal builds a CFG for each method, where basic blocks are structured as sequences of GraalIR nodes. Branch instructions at the IR level directly map to the edges of the CFG, establishing a clear link between data-driven operations and the program's execution flow.

2.3 ML background

Supervised machine learning [27, 28] is a widely used approach in which a model learns to associate input features with corresponding output labels by training on a labeled dataset. In classification problems, each data instance is expressed as a feature vector, and the model's objective is to accurately assign new, unseen instances to one of several predefined categories. Before training, datasets typically undergo preprocessing steps, such as feature standardization and techniques to address class imbalance, including instance weighting, oversampling, or undersampling. Once trained, the model's effectiveness is assessed using performance metrics such as accuracy, precision, recall, and the F1 score, together providing a comprehensive view of predictive quality [29].

Tree-based models [30] are well-suited for classification problems involving structured, tabular data because of their resilience to noise, ability to model complex nonlinear feature interactions, and ease of interpretation. One of the most powerful and widely adopted methods in this category is XGBoost (Extreme Gradient Boosting) [31, 32], a scalable and highly efficient boosting algorithm that sequentially constructs an ensemble of decision trees. XGBoost offers several advanced features, including native support for missing values, and incorporates both L1 (Lasso) and L2 (Ridge) regularization techniques [33, 34] to mitigate the risk of overfitting. Additionally, it is designed for high performance, enabling fast and parallelized model training on large datasets [35].

One of XGBoost's key advantages is its ability to offer detailed insights into feature importance using metrics such as gain [33]. Gain quantifies the improvement in the model's loss function achieved by splitting on a particular feature, with higher gain values indicating that the feature has a greater impact on the model's predictions. This interpretability helps identify which features most strongly influence the classification outcome, offering both diagnostic value and opportunities for feature engineering. To further enhance model performance, hyperparameter tuning is typically performed using grid search [36], which systematically explores combinations of parameters and selects the best configuration based on validation performance.

3 CFG Characterization and Feature Extraction

We integrate the feature extraction and performance measurement pipeline directly into the Graal compiler infrastructure by introducing a custom phase into the method compilation queue. This newly added phase is designed to carry out configurable experiments during compilation, enabling it to extract structural features from the CFG and evaluate the performance of a specific graph traversal algorithm. By embedding this functionality within the compiler pipeline, we ensure minimal disruption to the existing workflow while enabling systematic data collection and analysis at scale.

To train the machine learning model to learn the connection between a control flow graph's structure and the most efficient traversal algorithm, we defined a set of 24 descriptive features. These features, summarized in **Table 1**, are designed to capture key structural aspects of CFGs, including their size, shape, branching complexity, and connectivity characteristics. The rationale behind this selection is grounded in the observation that the performance of traversal algorithms, such as BFS and DFS, is closely influenced by these properties [37]. By quantifying these elements, the model can learn patterns that generalize across different types of CFGs, enabling it to predict the optimal traversal strategy for new, unseen graphs.

To identify whether BFS or DFS is the more efficient traversal algorithm for a given method's CFG, we measured and compared the average execution times of both algorithms over 100 iterations. Each CFG was labeled according to the traversal method with the lower average execution time. To ensure accurate and consistent timing measurements, we applied a range of system-level optimizations: Intel Turbo Boost was disabled, CPU C-states were set to 0, and the CPU frequency scaling governor was fixed to *performance* mode. Furthermore, to minimize background interference and context switching, we disabled Hyper-Threading and increased the priority of the benchmarking process.

4 Dataset

To train machine learning models for predicting the optimal traversal algorithm for control flow graphs, we constructed a labeled dataset of 221,749 graphs [20]. CFGs were extracted from programs in the DaCapo [9] and Renaissance [10] benchmarking suites. Renaissance includes modern JVM-based workloads built on frameworks like Akka, Spark, and ScalaSTM, while DaCapo represents traditional Java applications from domains such as web services, cryptography, and data analysis. Together, they comprise a dataset used to benchmark the GraalVM compiler and provide a diverse set of CFGs in terms of depth, width, connectivity, and branching complexity. All benchmarks are executed in multiple iterations, where each iteration corresponds to a complete execution of a program. Including both Renaissance and DaCapo allowed us to collect CFGs from a broad spectrum of workloads, combining contemporary and legacy codebases. This diversity supports the training and evaluation of machine learning models for predicting traversal strategies across different types of graph structures.

Table 1
CFG features contained in the dataset.

Feature Name	**Description**
V	Number of nodes in the CFG
E	Number of edges in the CFG
Depth	Maximal DFS traversal recursion depth
Width	Maximal number of nodes in the BFS queue during the CFG traversal
Binary splits	Number of nodes with exactly two successors
Non-binary splits	Number of nodes with more than two successors
Total splits	Number of nodes with more than one successor
The ratio of nodes to edges	V/E; Ratio of vertices to edges, quantifying graph sparsity
The ratio of edges to nodes	E/V; Ratio of edges to vertices, quantifying average connectivity per node
Min. degree	Minimal total degree of a CFG node, including both ingoing and outgoing edges
Max. degree	Maximal total degree of a CFG node, including both ingoing and outgoing edges
Avg. degree	The average total degree of a CFG node, including both ingoing and outgoing edges
Coeff. variation degrees	Relative variability of nodes' total degrees, indicating dispersion in connectivity
The entropy of the distribution degrees	The randomness in nodes' total degrees, indicating connectivity heterogeneity
Min. in-degree	Minimal number of predecessors of a CFG node
Max. in-degree	Maximal number of predecessor edges of a CFG node
Avg. in-degree	Average number of predecessors of a CFG node
Coeff. variation in-degrees	Relative variability of nodes' in-degrees, showing heterogeneity in incoming-edge distribution
The entropy of the distribution in-degrees	Randomness in incoming-edge distribution
Max. out-degree	Maximal number of outgoing edges of a CFG node
Coeff. variation out-degrees	Relative variability of nodes' out-degrees, showing heterogeneity in incoming-edge distribution
The entropy of the distribution out-degrees	Randomness in outgoing-edge distribution
Avg. split cardinality (including binary splits)	Average number of successors of nodes with more than one successor
Avg. split cardinality (excluding binary splits)	Average number of successors of nodes with more than two successors

Running benchmarks in iterations enables a more accurate estimation of the average execution time by aggregating results across repeated program executions. Each CFG was processed within the GraalVM compiler infrastructure using a custom phase that extracted structural graph features and measured traversal performance for both BFS and DFS. Traversal times were recorded across 100 iterations and aggregated using statistics such as mean, standard deviation, quartiles, and median. Each dataset instance includes the CFG features, the traversal strategy, aggregated performance metrics, and a label indicating which strategy was faster on average.

Fig. 3 illustrates the process of constructing the dataset, and it involves the following steps:

1. **Benchmark Preparation** — We organize the benchmark sources by suite and then by individual benchmark to support structured, batch-based processing in subsequent steps. Each benchmark is placed in a predefined directory structure, enabling uniform handling during compilation, control flow graph extraction, feature collection, and traversal measurements.
2. **Parsing and CFG construction** — Each benchmark program is compiled using the GraalVM compiler. During compilation, a control flow graph is constructed for each method.
3. **Traversal Measurement Phase** — A custom compiler phase is inserted into the Graal compilation pipeline to perform traversal measurements on each CFG. For every CFG:
    - Features of the CFG are extracted: *f* = *extractFeatures*(*c*).
    - Each traversal strategy $t \in$ {DFS, BFS} is applied.
    - The traversal times are measured over 100 iterations.
    - For each iteration, we record its traversal time. Additionally, we record the traversal strategy and CFG features to facilitate the aggregation processes in the following steps.
4. **Aggregation of Measurements** — For each CFG and traversal strategy, we aggregate traversal times measured over 100 iterations using the following statistical descriptors:
    - quartiles (Q1, Q2, Q3),
    - minimum and maximum values,
    - mean and standard deviation,
    - median and other summary statistics.
5. **Label Assignment and Dataset Entry Creation** — Each CFG is represented by a data entry for each traversal algorithm. These entries consist of:
    - The extracted features of the CFG
    - The traversal type (DFS or BFS)
    - Aggregated statistics of the traversal time
    - A label indicating which traversal strategy was faster on average.

Final Dataset — The final dataset contains over 221,000 entries. Each entry corresponds to a unique pair of a CFG and a traversal algorithm, labeled with performance data and graph features. This dataset enables the training of machine learning models to predict the optimal traversal algorithm based on features of the CFG. Label distribution is shown in Fig. 4. The dataset is open and published on Zenodo [20].

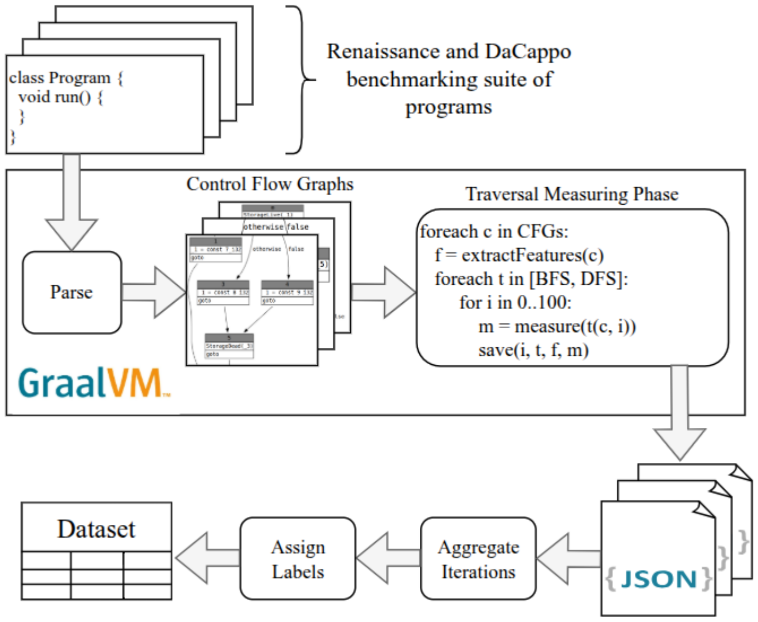

Fig. 3 – *Dataset creation pipeline.*

Table 2 provides summary statistics for each of the 24 features, including the mean, standard deviation, and key quartile values, offering insight into the overall distribution and variability of the dataset. The control flow graphs in the dataset exhibit considerable structural diversity. The number of nodes of the graph ranges from as few as 1 to as many as 5,716, while the number of edges spans from 0 to 8,571. On average, graphs contain 17.78 nodes and 23 edges. Median values for both the depth and the width of the graph are around 3, suggesting that most CFGs are relatively shallow and narrow. The majority of branching points involve binary splits, with very few branches having more than two splits, leading to an average of 7.26 splits per graph. Although the average node degree is low (1.61), some graphs include highly connected nodes, with maximum in- or out-degrees reaching as high as 2,857. Additionally, features related to degree variability and entropy indicate a broad range of structural complexity. This level of heterogeneity is essential for training a machine learning model that can generalize well across diverse graph topologies.

Table 2
Summary statistics for extracted CFG features.

Feature Name	Mean	Std	Min	25%	50%	75%	Max
V	17.78	55.58	1.00	3.00	6.00	16.00	5,716.00
E	23.00	80.42	0.00	2.00	6.00	19.00	8,571.00
Depth	6.14	18.34	1.00	1.00	3.00	6.00	2,857.00
Width	3.29	4.00	1.00	2.00	3.00	4.00	735.00
Binary splits	7.24	25.28	0.00	1.00	2.00	6.00	2,857.00
Non-binary splits	0.02	0.17	0.00	0.00	0.00	0.00	18.00
Total splits	7.26	25.31	0.00	1.00	2.00	6.00	2,857.00
The ratio of nodes to edges	0.75	0.49	0.00	0.69	0.80	1.00	1.50
The ratio of edges to nodes	0.82	0.51	0.00	0.67	1.00	1.25	1.98
Min. degree	0.85	0.56	0.00	1.00	1.00	1.00	2.00
Max. degree	5.93	21.52	0.00	1.00	3.00	5.00	2,857.00
Avg. degree	1.61	1.06	0.00	1.00	2.00	2.55	3.96
Coeff. variation degrees	0.30	0.36	0.00	0.00	0.31	0.41	12.59
The entropy of the distribution degrees	0.87	0.74	0.00	0.00	1.33	1.48	2.16
Min. in-degree	0.76	0.43	0.00	1.00	1.00	1.00	1.00
Max. in-degree	5.70	21.55	0.00	1.00	2.00	5.00	2,857.00
Avg. in-degree	0.94	0.55	0.00	1.00	1.20	1.33	1.98
Coeff. variation in-degrees	0.54	0.74	0.00	0.00	0.33	0.80	25.19
The entropy of the distribution in-degrees	0.34	0.32	0.00	0.00	0.37	0.61	1.10
Max. out-degree	1.58	1.78	0.00	2.00	2.00	2.00	278.00
Coeff. variation out-degrees	0.60	0.46	0.00	0.37	0.57	0.82	11.62
The entropy of the distribution out-degrees	1.01	0.61	0.00	0.92	1.27	1.52	2.02
Avg. split cardinality (including binary splits)	0.09	1.52	0.00	0.00	0.00	0.00	232.00
Avg. split cardinality (excluding binary splits)	1.52	0.93	0.00	2.00	2.00	2.00	94.67

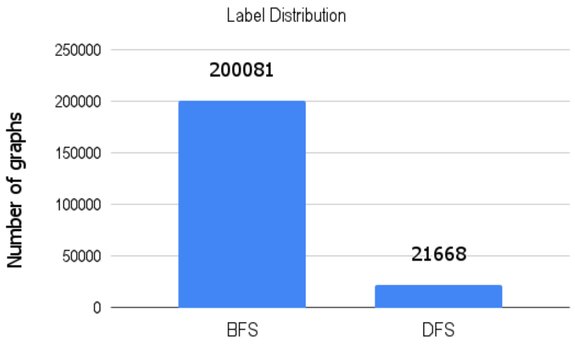

Fig. 4 – *Distribution of labels in the dataset.*

5 Model Training and Implementation

To establish the model training and evaluation pipeline, we randomly divided the dataset into two subsets, with 80% of the data allocated for model training. In contrast, the remaining 20% was reserved for testing its performance. Given the high class imbalance, we apply instance weighting to penalize misclassifications of underrepresented instances more heavily. This ensures that the distribution of the optimal traversal algorithm (BFS or DFS) remains consistent across the training and test sets, which is important for reliable model evaluation.

We selected XGBoost as the machine learning model for predicting the optimal traversal algorithm for each CFG. This choice was motivated by XGBoost's strong performance on structured, tabular data, as well as its efficiency and ability to model complex, nonlinear feature interactions. Beyond predictive accuracy, XGBoost also offers valuable interpretability through feature importance metrics, enabling us to understand which graph features most significantly influence the model's decisions. These characteristics make it particularly well-suited for our use case, where both performance and insight into the decision process are important.

To identify the best hyperparameters for the XGBoost model, we performed a grid search combined with 5-fold cross-validation on the training dataset. The weighted F1 score was used as the evaluation metric, as it accounts for class imbalance and ensures balanced predictive performance across both classes. We performed a grid search over two key hyperparameters: maximum tree depth and ensemble size. We tested tree depths of 10, 20, and 30, along with ensemble sizes of 500, 1,000, 2,000, 3,000, and 5,000 trees. The configuration that achieved the highest cross-validated performance consisted of 2,000 trees, each of depth 20, striking a good balance between model expressiveness and the risk of overfitting.

This optimal setting was then used to train the final model on the full training dataset.

To handle a class imbalance in the dataset, we employed instance weighting during the model training. Multiple models were trained using different weighting strategies to evaluate their effect on classification performance. In the first training configuration, instance weights were automatically adjusted based on the class distribution in the training data, ensuring proportional contribution from both the BFS majority class and the less frequent DFS class. In other configurations, we manually assigned higher weights to the underrepresented DFS class to further amplify its influence during training. This strategy enabled us to assess how different weighting schemes affect model sensitivity and performance, particularly in correctly predicting the less common traversal algorithm.

6 Evaluation

In this section, we evaluate and compare different ML models. To do so, we compute and report standard evaluation metrics, including accuracy, precision, recall, and F1 score. In addition, we present confusion matrices for each model to highlight the nature and frequency of misclassifications. Finally, we perform a detailed analysis of the handcrafted feature set, examining which features contribute most to the model's predictive accuracy and how they influence classification outcomes.

6.1 Balanced instance weights

To address the class imbalance between the BFS and DFS labels, we train the models using a balanced instance weighting strategy. Specifically, we calculate class weights based on the inverse class frequency, scaled by the total number of training samples and the number of classes. The formula used to compute the weight w for class c, denoted as w_c, is given as: $w_c = N / (K * n_c)$, where N represents the total number of training samples, K represents the number of classes, and n_c represents the number of samples in class c. This approach ensures that classes with fewer samples receive higher weights, thereby penalizing their misclassifications more heavily during training. In our case, the BFS class, being more frequent, receives a weight of approximately 0.55, while the less frequent DFS class is assigned a significantly higher weight of around 5.12.

The large gap between class weights (0.55 for BFS vs. 5.12 for DFS) reflects the training dataset's class imbalance. BFS, as the majority class, is down-weighted, while the rarer DFS class is boosted to ensure balanced learning; in a balanced dataset, both weights would be approximately equal. This weighting ensures that the model places more emphasis on correctly learning patterns associated with the underrepresented DFS class, improving overall classification balance and reducing bias toward the majority class.

On the test set, which contains 44,350 instances, the XGBoost model achieves an overall accuracy of 78.25%. The model achieved 89.44% precision, 78.25% recall, and an F1 score of 82.10%. These results indicate strong predictive performance, particularly for the majority class. However, the relatively lower recall value indicates that the model struggles more with correctly identifying DFS instances, reflecting the underlying class imbalance and the greater difficulty in learning patterns associated with the less frequent class. Nonetheless, the F1 score demonstrates a reasonable trade-off between precision and recall.

A more detailed view of the model's classification performance for each class is presented in **Table 3**. To offer a comprehensive assessment of the model's effectiveness, we report both macro and weighted averages of the standard evaluation metrics. The macro average, which treats each class equally regardless of its frequency, yields a precision of 0.62, a recall of 0.75, and an F1 score of 0.63. In contrast, the weighted average, which accounts for the class distribution in the dataset, reports a precision of 0.89, a recall of 0.78, and an F1 score of 0.82. These results highlight the model's strong performance on the majority class (BFS) while also revealing room for improvement in detecting the less common DFS class. The gap between macro and weighted scores highlights the effect of class imbalance.

The confusion matrix for the XGBoost model on the test set, presented in Fig. 5, provides insight into how the model performs across the two classes. Among the 40,016 instances labeled as BFS, 8,400 were incorrectly classified as DFS, resulting in a misclassification rate of 21.0% for the BFS class. For the DFS class, which comprises 4,334 instances, the model misclassified 1,244 as BFS, resulting in a higher misclassification rate of 28.7%. These results indicate that while the model is relatively accurate overall, it has more difficulty correctly identifying DFS instances, which is consistent with the class imbalance observed in the dataset.

It is important to emphasize that the XGBoost model does not exhibit signs of overfitting to the training data. On the training set, the model achieves an accuracy of 79.28%, a precision of 90.63%, a recall of 79.28%, and an F1 score of 82.99%. These values are very close to the corresponding metrics on the test set, with only marginal improvements. This consistency between training and test performance indicates that the model generalizes well and maintains stable predictive behavior across unseen data, further confirming the robustness of the chosen feature set and model configuration.

The presented results demonstrate that the model performs well overall, effectively capturing the majority of instances across both classes. While there is still room for improvement, particularly in reducing misclassifications within the underrepresented DFS class, the model demonstrates strong generalization capabilities and effectively handles class imbalance. These findings suggest that the chosen features and training strategy are effective, though further refinements could enhance performance, particularly for the minority class.

Table 3
Per-class classification report on test set for the XGBoost ML model consisting of 2000 trees with maximal depth of 20.

Class	*Precision*	*Recall*	*F1 Score*	*No. of instances*
BFS	0.96	0.79	0.87	40,016
DFS	0.27	0.71	0.39	4,334

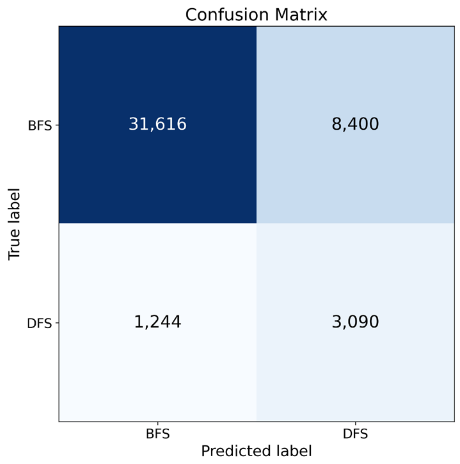

Fig. 5 – *Confusion matrix on the test set for the XGBoost model with 2,000 trees (max depth 20) trained using balanced class weights.*

6.2 Feature importance

The XGBoost model not only offers strong predictive performance but also yields valuable insights into the effectiveness of the handcrafted features used to characterize CFG blocks. In Fig. 6, we present the most influential features, as determined by their gain, a metric that quantifies each feature's contribution to reducing the model's overall loss during training. Features with higher gain values have a more significant impact on guiding the model's decision-making process. This analysis helps identify which structural aspects of the CFGs are most relevant for predicting the optimal traversal strategy, offering guidance for future feature engineering and model improvements.

Among all the evaluated features, the number of nodes emerges as the most dominant, achieving a gain of 3.12, which far surpasses that of all other features. This indicates that the overall size of a CFG is the most influential factor in the model's prediction of the optimal traversal algorithm. The next most impactful features — average in-degree (0.24), number of non-binary splits (0.24), and the node-to-edge ratio (0.20) — exhibit substantially lower gains. This disparity suggests that size-related characteristics, particularly node count, play a central role in the model's decision-making process. Nonetheless, a range of other features, including entropy measures and various degree-based metrics, provide moderate contributions. They enrich the model's understanding by capturing more subtle aspects of graph structure. Overall, the distribution of gain values indicates that while the model relies heavily on coarse-grained structural properties, it also leverages more detailed topological and statistical descriptors to refine its predictions.

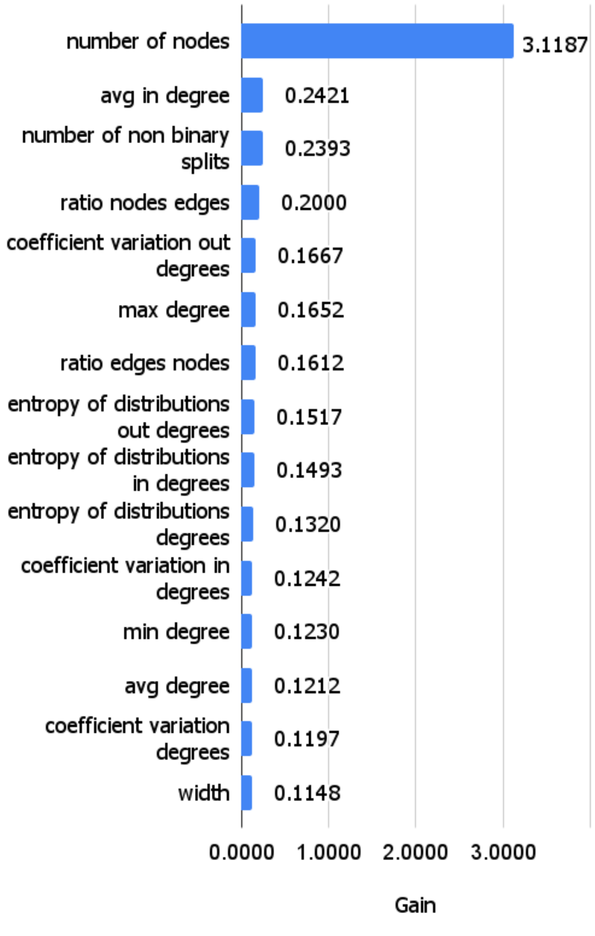

Fig. 6 – *Feature gains derived from the trained XGBoost ensemble.*

6.3 Improving DFS misclassifications

To enhance the model's sensitivity to the underrepresented DFS class, we experimented with manually defined instance weights during training. This strategy aimed to reduce the misclassification rate for DFS by explicitly increasing its influence in the learning process. In one such configuration, we assigned a weight of 0.5 to BFS instances and 10.0 to DFS instances - effectively placing approximately 1.86 times more emphasis on DFS. As a result, the confusion matrix for this model, shown in Fig. 7, revealed an improved balance between the classes, with a misclassification rate of 24.88% for BFS and 26.40% for DFS. This represents a noticeable improvement in DFS classification compared to the baseline, suggesting that targeted weighting can help mitigate the effects of class imbalance and lead to more equitable performance across classes.

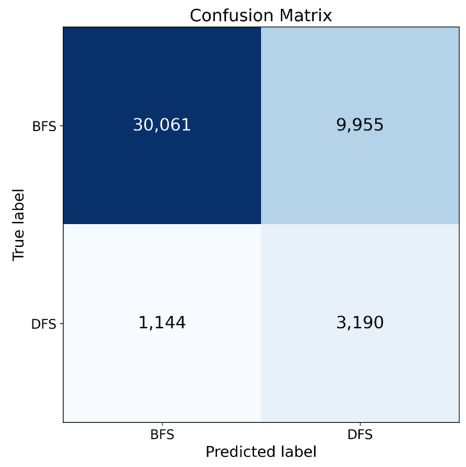

Fig. 7 – *Confusion matrix on the test set for the XGBoost model with 2,000 trees (max. depth of 20) trained using class weights BFS: 0.5 and DFS 10.0.*

As expected, emphasizing the underrepresented DFS class during training leads to a slight reduction in overall classification metrics, particularly those that treat all classes equally. However, the decrease is not substantial and reflects a trade-off aimed at achieving a better balance. The model trained with manually assigned weights favoring DFS instances achieves the following unweighted performance on the test set: an accuracy of 74.97%, a precision of 89.29%, a recall of 74.97%, and an F1 score of 79.73%. These results indicate that, while overall performance drops slightly, the model becomes more equitable across classes without significantly compromising predictive power.

When comparing these results to those of the original model, it becomes clear that assigning higher weights to DFS instances results in a modest decrease in overall accuracy and F1 score. However, precision remains nearly unchanged, indicating that the model maintains a high proportion of correct positive predictions. The trade-off between accuracy and recall highlights the effect of prioritizing the minority DFS class, resulting in a slight reduction in BFS misclassification.

6.4 Simplified model

To examine how model complexity affects performance, we trained smaller XGBoost models by reducing both the number of trees and the maximum depth of each tree. The most effective configuration in this simplified setup comprises 100 trees, each with a maximum depth of 3 (the confusion matrix is shown in Fig. 8). Despite its reduced complexity, this model achieved strong results on the test set: accuracy of 79.59%, precision of 89.47%, recall of 79.59%, and an F1 score of 83.05%. The misclassification rates were 19.35% for BFS and 30.11% for DFS.

Interestingly, these metrics outperform those of the larger ensemble model, which has 2,000 trees and a depth of 20. However, this improvement is not uniform across classes. While the simpler model reduces the BFS misclassification rate by 1.65%, it increases the DFS misclassification rate by 1.4%, highlighting the trade-off between model simplicity and balanced class performance.

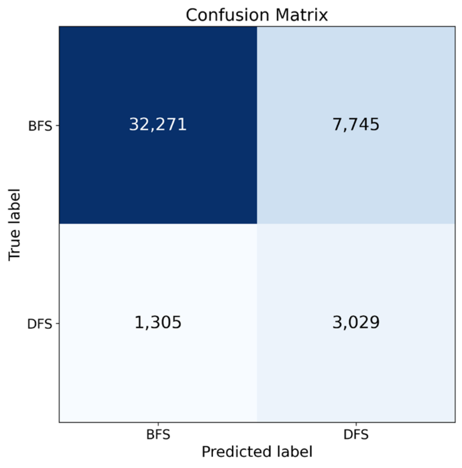

Fig. 8 – *Confusion matrix on the test set for the XGBoost model with 100 trees (max. depth of 3) trained using balanced class weights.*

6.5 Discussion

To the best of our knowledge, this specific problem has not been addressed in prior work. As a result, an appropriate baseline for comparison is a simple model that makes predictions by randomly selecting a class, either with uniform probability or with a biased distribution that reflects the dataset's imbalance (e.g., 0.8 for BFS and 0.2 for DFS). Additionally, given the highly imbalanced nature of the dataset, it is also meaningful to compare our trained models to a trivial baseline that always predicts the majority class (BFS). These baselines help contextualize the performance of our approach and demonstrate that the XGBoost models offer significant improvements over naive or uninformed prediction strategies.

Our models significantly outperformed this baseline. An XGBoost model with 2,000 trees and maximum depth 20 achieved misclassification rates of 21% for BFS and 28.7% for DFS. A model with 100 trees and maximum depth 3 reached 19.35% for BFS and 30.11% for DFS. Another model with 2,000 trees, depth 20, and DFS-weighted training produced balanced misclassification rates of 24.88% for BFS and 26.40% for DFS, showing a clear improvement over random baselines.

In a dataset where 80% of instances belong to the majority class and 20% to the minority class, a random classifier with uniform class selection achieves an overall accuracy of 50%. For the minority class, it yields 9% precision, 50% recall, and an F1 score of 15%. For the majority class, the precision is 91%, the recall is 50%, and the F1 score is 65%. The confusion matrix for this classifier indicates a 50% misclassification rate for both classes, underscoring its limited ability to make accurate predictions.

A random classifier that selects the majority class 80% of the time and the minority class 20% of the time achieves a higher overall accuracy of 68%. For the minority class, both precision and recall drop to 20%, resulting in an F1 score of 20%. In contrast, the majority class sees 80% precision and recall, with an F1 score of 80%. However, this approach misclassifies 80% of the minority instances, highlighting its ineffectiveness in handling class imbalance. These results illustrate the limitations of naive or probabilistic classifiers, particularly in imbalanced settings.

A classifier that always predicts the majority class achieves 80% accuracy, but this is misleading. It completely fails in the minority class, with 0% precision, recall, and F1 score. For the majority class, all three metrics are 100%. The confusion matrix confirms 0% misclassification for the majority class and 100% for the minority class, illustrating the model’s inability to generalize beyond the dominant class.

Our models substantially outperformed these baseline classifiers. The XGBoost model with 2,000 trees and a maximum depth of 20 achieved misclassification rates of 21.0% for BFS and 28.7% for DFS, demonstrating strong performance across both classes. A more compact model with 100 trees and a maximum depth of 3 performed comparably, with 19.35% misclassification for BFS and 30.11% for DFS, highlighting the effectiveness of even low-complexity models. Additionally, a variant trained with DFS-weighted instance balancing (2,000 trees, depth 20) achieved more balanced misclassification rates of 24.88% for BFS and 26.40% for DFS. These results clearly demonstrate that all trained XGBoost models substantially outperform random or trivial baselines, especially in handling class imbalance, and offer meaningful accuracy improvements for both the majority and minority classes.

6.6 General applicability and end-to-end performance considerations

Although our models are optimized for programs targeting JVM languages such as Java or Scala, our approach is reusable and can be incorporated into other ecosystems, such as .NET or LLVM. .NET ecosystem provides support for similar capabilities as GraalVM, but targeting the set of languages operating on the Common Language Runtime and thus using Common Intermediary Language (CIL) instead of Java bytecode. Our feature-extraction and dataset creation pipeline can be injected as part of .NET NativeAOT compilation tool to extract features of method IR graphs. Similarly, LLVM is primarily oriented toward C/C++-like languages and utilizes a three-address based intermediate representation, from which pertinent features may be extracted. The proposed methodology may be integrated within the Clang compilation framework as a natural extension of its functionality.

End-to-end compilation time savings depend not only on the model's accuracy, but also on the time it takes for the ML model to make predictions (inference time). In addition, the estimation of the achievable time savings is further influenced by the size and shape of the project being compiled. Future research efforts should focus on refining the ML model in order to realize meaningful end-to-end compilation time savings. Such refinement may be pursued through the incorporation of novel and discriminative features, as well as through improvements in the quality, diversity, and representativeness of the underlying dataset.

7 Conclusions and Future Work

In this study, we investigated a set of control flow graph features to characterize graph structure and used them to train an ensemble-based XGBoost machine learning model capable of predicting the optimal traversal algorithm (BFS or DFS) for each graph. Through feature-importance analysis, we found that the number of nodes, average in-degree, and number of binary splits were among the most informative features for guiding traversal decisions. Importantly, these and most other high-impact features can be computed efficiently during the CFG construction phase without requiring an additional traversal. This makes the approach especially practical for use cases involving static or frequently reused graphs, such as those found in library code. The trained model achieves strong overall predictive performance, with balanced coverage of both classes. Although occasional misclassifications occur in the minority class, the model successfully mitigates the effects of class imbalance and demonstrates good generalization to unseen data.

In addition to using the XGBoost model for prediction, we leveraged it to rank the importance of our manually engineered input features. While handcrafted features offer interpretability and efficiency, an alternative direction involves automatically generating feature representations for each CFG block. Prior work in static profiling has explored the use of map-based features to describe CFG blocks [38, 39], offering a more flexible and potentially richer characterization. Building on this idea, we see an opportunity to extend our approach by incorporating automated feature extraction techniques. This could enable the application of more expressive machine learning models, such as deep neural networks or graph neural networks (GNNs), which are better suited for learning from complex graph structures. We plan to explore this direction in future work to improve prediction accuracy and model scalability further.

GNNs are particularly well-suited for modeling the structure of control flow graphs, as they operate directly on graph-structured data and can perform tasks such as graph-level binary classification. This makes GNNs a promising alternative to the XGBoost-based model explored in this work. However, unlike tree-based models that rely on global graph features, GNNs require input features at the node level, where each node represents a basic block in the CFG. These node features can be derived from a subset of the informative features identified by XGBoost or from more flexible, map-based descriptors. Designing GNN architectures that are both expressive and computationally efficient is critical, given that GNNs typically incur higher inference times. Additionally, to fully leverage the potential of GNNs and improve their generalization, building a larger and more diverse dataset tailored for GNN training and evaluation will be an important step in the future.

In future work, we will focus on fine-tuning the traversals by modifying the data structures used in the algorithms and by combining different traversal strategies. Additionally, we will investigate how the results extend to traversal methods beyond BFS and DFS. We aim to explore the benefits of integrating the best-performing models into the Oracle GraalVM Native Image compiler.

8 Acknowledgments

This work is supported by the Ministry of Science, Technology and Innovation of the Republic of Serbia, under agreement No. 451-03-47/2023-01/200104, as well as through the research project provided by Oracle America, Inc.

9 References

[1] O. Ore: Graphs and their Uses, Mathematical Association of America, Washington, 1990.

[2] K. Bobrovnikova, S. Lysenko, B. Savenko, P. Gaj, O. Savenko: Technique for IoT Malware Detection Based on Control Flow Graph Analysis, Radioelectronic and Computer Systems, No. 1, 2022, pp. 141 – 153.

[3] Z. Ma, H. Ge, Y. Liu, M. Zhao, J. Ma: A Combination Method for Android Malware Detection Based on Control Flow Graphs and Machine Learning Algorithms, IEEE Access, Vol. 7, January 2019, pp. 21235 – 21245.

[4] D. Bruschi, L. Martignoni, M. Monga: Detecting Self-Mutating Malware Using Control-Flow Graph Matching, Proceedings of the 3rd International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Berlin, Germany, July 2006, pp. 129 – 143.

[5] F. E. Allen: Control Flow Analysis, ACM SIGPLAN Notices, Vol. 5, No. 7, July 1970, pp. 1 – 19.

[6] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, M. Wolczko: One VM to Rule them All, Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Indianapolis, USA, October 2013, pp. 187 – 204.

[7] GraalVM: Build Faster, Smaller, Leaner Applications, Available at:
https://www.graalvm.org/

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: Compilers-Principles, Techniques and Tools, 2nd Edition, Pearson Education, Inc., Boston, San Francisco, New York, 2007.

[9] S. M. Blackburn et al.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis, ACM SIGPLAN Notices, Vol. 41, No. 10, October 2006, pp. 169 – 190.

[10] A. Prokopec et al., Renaissance: Benchmarking Suite for Parallel Applications on the JVM, Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, Phoenix, USA, June 2019, pp. 31 – 47.

[11] A. Kanuparthi, J. Rajendran, R. Karri: Controlling Your Control Flow Graph, Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, USA, May 2016, pp. 43 – 48.

[12] J. Wang, C. Zhang, L. Chen, Y. Rong, Y. Wu, H. Wang, W. Tan, Q. Li, Z. Li: Improving ML-Based Binary Function Similarity Detection by Assessing and Deprioritizing Control Flow Graph Features, Proceedings of the 33rd USENIX Security Symposium, Philadelphia, USA, August 2024, pp. 4265 – 4282.

[13] Z. Zhao: A Virus Detection Scheme Based on Features of Control Flow Graph, Proceedings of the 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), Deng Feng, China, August 2011, pp. 943 – 947.

[14] M. Čugurović, I. Ristović, S. Stanojević, M. Spasić, V. Marinković, M. Vujošević Janičić: ML-Driven Prediction of Optimal Control Flow Graph Traversal Algorithm in Modern Applications, Proceedings of the 12th International Conference on Electrical, Electronic and Computing Engineering (IcETRAN), Čačak, Serbia, June 2025, pp. 1 – 6.

[15] F. R. K. Chung: Spectral Graph Theory, American Mathematical Society, Providence, 1997.

[16] Q. Sun, E. Abdukhamidov, T. Abuhmed, M. Abuhamad: Leveraging Spectral Representations of Control Flow Graphs for Efficient Analysis of Windows Malware, Proceedings of the ACM

on Asia Conference on Computer and Communications Security, Nagasaki, Japan, May 2022, pp. 1240 – 1242.

[17] S. Mitra, S. A. Torri, S. Mittal: Survey of Malware Analysis through Control Flow Graph Using Machine Learning, Proceedings of the IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Exeter, UK, November 2023, pp. 1554 – 1561.

[18] R. Mercado, E. J. Bjerrum, O. Engkvist: Exploring Graph Traversal Algorithms in Graph-Based Molecular Generation, Journal of Chemical Information and Modeling, Vol. 62, No. 9, May 2022, pp. 2093 – 2100.

[19] I. Ristović, M. Čugurović, S. Stanojević, M. Spasić, V. Marinković, M. Vujošević Janičić: Efikasan obilazak grafova kontrole toka, Proceedings of the 30th National Conference on Information and Communication Technologies (YU INFO), Kopaonik, Serbia, March 2024, pp. 89 – 94.

[20] M. Spasić, S. Stanojević, I. Ristović, M. Čugurović, V. Marinković, M. Vujošević Janičić: Control Flow Graphs (CFGs) for JVM Applications/Benchmarks Compiled with GraalVM Native Image, Zenodo, July 2025, version v2.

[21] K. Zhu, Y. Lu, H. Huang, L. Yu, J. Zhao: Constructing More Complete Control Flow Graphs Utilizing Directed Gray-Box Fuzzing, Applied Sciences, Vol. 11, No. 3, February 2021, p. 1351.

[22] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, H. Mössenböck: An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler, Proceedings of the 7th ACM Workshop on Virtual machines and Intermediate Languages, Indianapolis, USA, October 2013, pp. 1 – 10.

[23] C. Wimmer, C. Stancu, P. Hofer, V. Jovanović, P. Wögerer, P. B. Kessler, O. Pliss, T. Würthinger: Initialize Once, Start Fast: Application Initialization at Build Time, Proceedings of the ACM on Programming Languages, Vol. 3, No. OOPSLA, October 2019, pp. 184.

[24] R. Bruno, S. Ivanenko, S. Wang, J. Stevanović, V. Jovanović: Graalvisor: Virtualized Polyglot Runtime for Serverless Applications, arXiv:2212.10131v1 [cs.DC], December 2022, pp. 1 – 17.

[25] GraalOS: High-Performance Serverless Application Deployment Platform, Available at: https://graal.cloud/graalos/

[26] C. Click, M. Paleczny: A Simple Graph-Based Intermediate Representation, ACM SIGPLAN Notices, Vol. 30, No. 3, March 1995, pp. 35 – 49.

[27] C. Bishop, Nasrabadi, M. Nasser, Pattern Recognition and Machine Learning, Information Science and Statistics, Springer New York, NY, 2006, pp. 179 – 181.

[28] T. Hastie, R. Tibshirani, J. Friedman: Model Inference and Averaging, Ch. 3, The Elements of Statistical Learning, 1st Edition, Springer, New York, 2001.

[29] M. Sokolova, G. Lapalme: A Systematic Analysis of Performance Measures for Classification Tasks, Information Processing & Management, Vol. 45, No. 4, July 2009, pp. 427 – 437.

[30] L. Breiman, M. Last, J. Rice: Random Forests: Finding Quasars, Ch. 16, Statistical Challenges in Astronomy, 1st Edition, Springer, New York, 2003.

[31] C. Bentéjac, A. Csörgő, G. Martínez-Muñoz: A Comparative Analysis of Gradient Boosting Algorithms, Artificial Intelligence Review, Vol. 54, 2021, pp. 1937 – 1967.

[32] T. Chen, C. Guestrin: XGBoost: A Scalable Tree Boosting System, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16), San Francisco, CA, USA, August 13-17, 2016, 785 – 794.

[33] A. E. Hoerl, R. W. Kennard: Ridge Regression: Biased Estimation for Nonorthogonal Problems, Technometrics, Vol. 12, No. 1, February 1970, pp. 55 – 67.

[34] R. Tibshirani: Regression Shrinkage and Selection Via the Lasso, Journal of the Royal Statistical Society, Series B: Statistical Methodology, Vol. 58, No. 1, January 1996, pp. 267 – 288.

[35] D. Nielsen: Tree Boosting with XGBoost – Why Does XGBoost Win „Every“ Machine Learning Competition?, MSc Thesis, Norwegian University of Science and Technology, Trondheim, 2016.

[36] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, D. D. Cox: Hyperopt: A Python Library for Model Selection and Hyperparameter Optimization, Computational Science & Discovery, Vol. 8, No. 1, January 2015, p. 014008.

[37] C. S. M. Ali, I. M. Ibrahim: A Review of Graph Traversal Algorithms: Techniques and Applications in Network Analysis, Asian Journal of Research in Computer Science, Vol. 18, No. 3, February 2025, pp. 61 – 72.

[38] M. Čugurović, M. Vujošević Janičić, V. Jovanović, T. Würthinger: GraalSP: Polyglot, Efficient, and Robust Machine Learning-Based Static Profiler, Journal of Systems and Software, Vol. 213, July 2024, p. 112058.

[39] L. Milikić, M. Čugurović, V. Jovanović: GraalNN: Context-Sensitive Static Profiling with Graph Neural Networks, Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization, Las Vegas, USA, March 2025, pp. 123 – 136.