POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Performance estimation of obfuscated applications with software metrics

Supervisors:

Prof. Cataldo Basile

Dott. Daniele Canavese

Candidate:

Francesco Sattolo

Academic Year 2023/2024
Torino

Abstract

Software companies that develop and distribute any application need to constantly face the challenge of an adversary manipulating it to alter its behavior, stealing intellectual property such as proprietary algorithms and distributing illegal copies. To limit the economic losses caused by such malicious activities, various protection techniques were invented to increase the time and cost required by the attackers to succeed. The first line of defense is given by obfuscation techniques, which aim to make reverse engineering the code difficult at the expense of execution performance. Given the high amount of existing techniques and the challenge of evaluating protection strength and performance overhead, finding the best tradeoff between these two objectives is a complex process. Currently, this is done by manually applying and testing different techniques, resulting in long development times or suboptimal choices. The main goal of this project is to investigate whether machine learning can simplify this process by accurately predicting the performance overhead of an obfuscation technique. This would allow developers to make informed decisions more quickly, reducing development costs and resulting in faster and more secure applications. The first phase consisted of finding precise ways to measure low-level application performance metrics while minimizing sources of indeterminism. In the second phase, the focus was on collecting static metrics at the code level. A framework was created to collect these static and dynamic metrics before and after applying various obfuscation techniques to functions. The resulting dataset was then used to train machine learning models to predict the performance degradation of applying an obfuscation technique to a given function.

Ringraziamenti

Ora che questa tesi volge al termine, è il momento giusto per ripensare al lungo percorso che l'ha preceduta e che mi ha permesso di arrivare dove sono oggi.

Voglio esprimere tutta la mia gratitudine verso i miei genitori, che hanno sempre sostenuto le mie scelte senza farmi mancare niente, spronandomi, aiutandomi e interessandosi ai miei progressi.

Ringrazio anche tutti i membri della nostra (triplice) grande famiglia: i nonni, sempre interessati alla mia salute e al mio futuro per assicurarsi il mio benessere; Silvia, Paola, Gianni e gli zii, sempre disponibili a dare utili consigli e a scherzare insieme; Ilaria, Pietro e i cuginetti, che mi hanno fatto ricordare com'è crescere e quanta strada è passata, e per cui spero di diventare un esempio da seguire.

Voglio anche dedicare un saluto speciale a nonno Mariano, che con il suo modo di vivere e il suo sostegno rimarrà sempre con me, guardando questo traguardo da lassù, insieme a Pucci, che mi ha accompagnato durante tutto questo percorso di crescita.

Ringrazio Inchi e tutti i miei amici per essere stati sempre al mio fianco, avermi fatto stare bene e aver reso il viaggio divertente e pieno di vita.

Infine il grazie più grande va a te Eli, che più di tutti mi hai sopportato nei momenti di difficoltà e hai celebrato con me ogni successo. Questo percorso di studio non sarebbe stato lo stesso senza di te, e mentre ora finalmente giunge al termine, la nostra avventura non è che all'inizio.

Table of Contents

List of Figures

Chapter 1

Introduction

Ever since proprietary software existed, individuals have always tried to understand its inner workings, motivated by curiosity, personal challenge and recognition. As years passed and economic interests increased, companies looked for ways to implement Digital Rights Management (DRM) systems that would allow them to control the use, modification and distribution of their copyrighted works. Various categories of techniques were developed with this goal in mind: obfuscation to protect the intellectual property in the code, tamper-proofing to detect and respond to unauthorized code modifications by disrupting execution, watermarking to be able to track the original owner of unauthorized copies, birthmarking to detect illegal reuse of code fragments. Even with all these counter-measures in place, man-at-the-end (MATE) attacks, in which the attacker has unrestricted access to the software executable, environment and hardware on which it is being run, are so powerful that they will eventually succeed [1].

The widespread availability of unlawfully obtained software worldwide demonstrates the software's susceptibility to these attacks. According to the Business Software Alliance's latest reports, the commercial value of unlicensed software use in Europe alone between 2015 and 2017 exceeded 12 billion dollars. Depending on the country, unlicensed software installations ranged from 17% (Luxembourg) to 85% (Armenia) [2]. Additionally, data from the latest UK Online Copyright Infringement tracker surveys have confirmed a substantial increase in people downloading software from illegal sources, rising from 18% in 2019 to 38% in 2022 [3].

Given the inevitability of MATE attacks, the goal of obfuscation shifts to impeding attackers as effectively as possible. This would reduce the economic repercussions caused by pirated software, especially during the crucial post-launch window where revenue generation is most vulnerable. Increasing the effort the attackers require to understand the program, obfuscation delays the release of unauthorized cracked versions, providing a window for legitimate sales.

To achieve this, obfuscation techniques specifically target the reverse engineering process, altering the program's data, logic, or control flow. Estimating reliably the effort required to break all different protection techniques and their combinations is a very hard problem

because it is very difficult to model the behavior of every creative, motivated and skilled MATE attacker.

The modifications introduced by obfuscation have the side effect of impacting the application performance, as the final program will be more complex and will need to execute more operations.

While employing numerous complex and intrusive obfuscation techniques across the entire codebase would certainly slow down the attacker, this approach is not optimal. The significant performance degradation it causes would severely impact legitimate users, worsening the user experience and potentially harming software sales and public reputation. Therefore, developers that need to obfuscate their application need to find a better tradeoff between the strength of the protection and the performance impact on the application.

The current way of evaluating the performance overhead introduced by the various obfuscation techniques is to measure the performance of the normal and obfuscated code, and see how performance changed. This trial-and-error process can be time-consuming, slowing down software development and potentially resulting in suboptimal choices that offer less protection or slow down performance more than necessary.

This project aims to improve this process by providing the developers with a pre-trained machine-learning model that can accurately predict the performance overhead introduced by the obfuscation techniques. This would only require one extraction of the base application metrics to use as input, eliminating the need to obfuscate and measure performance for each technique repeatedly. This approach would allow developers to make informed decisions more quickly, reducing development costs and ultimately resulting in faster and more secure applications.

To achieve this, it was necessary to explore potential enhancements to the framework developed by Stefano Alberto and Antonio Licursi in their works "Towards the prediction of performance degradation of obfuscated code"[4], and "Data set generation for performance overhead prediction of obfuscated code" [5].

Several enhancements were made across the whole framework to create a higher quality and more representative dataset, as well as more accurate machine learning models. First, the application pool selection and input generation process were changed to increase representativeness and diversification in the dataset. Additionally, several new metrics, both static and dynamic, were selected to describe the program behaviour. New measurement strategies were implemented to ensure these dynamic features' accuracy. Subsequently, various strategies were explored to enhance the overall quality of the dataset. Finally, new machine learning models and new training strategies were tested to achieve the best possible performance prediction accuracy.

1.1 Chapters organization

The structure of the thesis is the following:

- Chapter 2, "Background and Related works": presents an overview of the technical aspects necessary to understand this thesis' topics. It also presents relevant works

used as foundations for this work.

- Chapter 3, “Problem statement and Design”: describes the thesis goal more formally and articulates the high-level design decisions made during the research process.
- Chapter 4, “Implementation”: describes in detail how each of the theoretical aspects described in the previous chapter were translated into a functional framework.
- Chapter 5, “Results”: presents and analyzes the results obtained from the implementation phase.
- Chapter 6, “Future works”: evaluates possible future improvements and extensions of the framework.
- Appendix A, “User manual”: explains how to use the framework.
- Appendix B, “Developer manual”: explains how to modify and expand the framework.
- Appendix C, “Events table”: list of events measured.

Chapter 2

Background and Related Works

This chapter presents relevant information to understand better the topics addressed within this thesis.

2.1 Obfuscation

Reverse engineering a program is the process of understanding its behavior, starting with little information about it. This can be used for unauthorized modifications, vulnerability exploitation or intellectual property theft in a man-at-the-end (MATE) scenario, where the attacker has unlimited access to the software executable and all the environment and the hardware it is being run on. In this situation, the attacker does not have the application's source code. Still, he can use various tools and techniques to obtain useful information from the binary file itself. Two of the most common tools are the disassembler, which translates binary code into assembly code and the decompiler, which tries to reconstruct the source code from the assembly code.

The primary goal of obfuscation techniques is to impede the reverse engineering process by modifying the code so that it remains difficult for an attacker to understand the application logic and underlying algorithms, even when utilizing tools such as disassemblers and decompilers. Crucially, these transformations are achieved without altering the program's intended functionality.

Obfuscation techniques can be applied at both the binary and source code levels. Binary obfuscation involves modifying the compiled executable, while source code obfuscation alters the original source code before compilation. In both cases, building an obfuscator from scratch is an incredibly complex task since it must be able to perform many sophisticated transformations to the application without altering its functionality.

For this reason, we decided to rely on one of the few publicly available obfuscators, Tigress [6]. It is a powerful and flexible source-to-source transformer that takes C source code as input and generates an obfuscated version of that code as output. It supports a wide range of transformation techniques that can be tailored using different parameters

and chained to build the desired level of protection. It can be further customized by specifying which functions must be obfuscated.

2.2 Dynamic features extraction

To measure and understand the performance overhead introduced by obfuscation, it is essential to understand what approaches we can use to analyze performance. Depending on the application and the analysis goals, we can measure performance at different levels of granularity. We can measure the performance of the whole application, the performance of a single function, or the performance of even smaller portions of the code. Since obfuscations are typically applied at the function level, focusing on performance analysis at this level makes sense. When we are only interested in the time taken by a function to execute, modern platforms let us access two different timers:

- The System clock: This is a system-wide monotonic timer with nanoseconds resolutions, which can be accessed using the `clock_gettime` Linux system call or from timing libraries in various programming languages.
- The Time Stamp Counter: This monotonic hardware timer counts the number of cycles elapsed. This can be accessed using the `__rdtsc()` compiler built-in function or the corresponding `RDTSC` assembly instruction. This is typically enough for developers who want a general idea of their code's performance. By instrumenting their code, timers can be queried to find how much time a function takes to execute.

In our case, we want to be able to predict a function's performance, so a deeper understanding of its performance behavior is needed. Hardware support is required to access these metrics. Performance Monitoring Counters (PMC) are hardware registers that can be configured to monitor a large set of performance metrics accurately and introduce low overhead. These metrics refer to architecture-specific events that can reveal deeper insights into the interaction of the application code with the hardware.

The Linux kernel includes the `perf` tool that can be used to access PMCs in different ways, using different commands:

- `perf stat` implements the Events Counting technique, reporting the total count of the specified events that occurred during the entire application execution. Another open-source tool based on `perf` is used to count events with function granularity: `perforator`.
- `perf record` implements the Events Sampling technique, which allows accessing event counters at regular intervals. This is commonly used by developers who want to explore the performance behavior of long repetitive workloads quickly. While this allows retrieving event counts with function granularity, its accuracy is inferior to the Event Counting technique.

2.3 Machine Learning

Machine learning is a subfield of artificial intelligence that focuses on developing statistical algorithms that can learn from data and generalize to unseen data, trying to optimize some performance criterion iteratively. In our case, we would like to obtain a model that leans from performance data collected from our functions and can generalize this information to predict performance overhead values for all other possible functions.

There are various ML algorithms categorized based on their learning approach. This thesis focuses on supervised learning, a specific category where the training data includes input features and output labels to predict. In this case, this label is represented by the performance overhead associated with the function and is available in the training data since we measured it in the previous steps. The model aims to learn the mapping between the features and the corresponding output.

Since the output we are trying to predict is not a categorical value, but a continuous value, we will use Regression algorithms instead of Classification algorithms.

Some of the most commonly used models in this category are:

- Decision Trees: This algorithm uses a tree structure to make predictions. The tree is built by splitting the data considering specific features at each node, ultimately resulting in leaf nodes representing the predicted output. Each node will represent part of the decision rule that will be used to predict the target variable for the new data. This technique guarantees high interpretability, but can be prone to overfitting.

- Random Forests: This algorithm builds multiple decision trees and uses the average of their predictions as the output. This slightly reduces interpretability in favor of less overfitting, higher accuracy and more stable results [7].

- Neural Networks: This technique uses interconnected layers of nodes (neurons) to process input data, each layer performing specific transformations. Learning is done by adjusting the weights of these connections based on the training data. They can learn complex patterns and relationships between features and outputs, making them highly flexible. The disadvantages of this technique are the requirement for extensive training and the fact that the results are challenging to interpret. [8]

2.4 Related works

"Towards the prediction of performance degradation of obfuscated code" by Stefano Alberto [4] and "Data set generation for performance overhead prediction of obfuscated code" by Antonio Licursi [5] are the theses upon which this project was built. They laid the foundations for the steps required and the difficulties that must be faced to obtain a working machine-learning model.

The specific problem addressed by this thesis is not found anywhere else in the literature. Some works can be found that partially overlap with specific the topics involved:

Denis Bakhvalov's book "Performance Analysis and Tuning on Modern CPUs" [9] covers numerous aspects of noise reduction and dynamic feature extraction techniques. This resource also provides a detailed overview of modern hardware architectures, and performance analysis approaches that can be used to achieve accurate performance measurements.

"Surreptitious software: obfuscation, watermarking, and tamper-proofing for software protection" is a book written by Christian Collberg's [10], one of the developers of the Tigress tool. It contains deep technical knowledge about everything related to software protection techniques, including obfuscation techniques, from both the attacker's and the defender's points of view.

Chapter 3

Problem Statement and Design

3.1 Problem Statement

This thesis aims to improve the framework created in the previous dissertations to predict obfuscation overhead more accurately. After a single initial feature extraction phase, the framework will allow application developers to choose

- which functions they want to obfuscate
- using which of the supported obfuscation techniques

and immediately obtain a prediction of the resulting performance overhead without needing to manually test each function and obfuscation technique. This would allow developers to make informed decisions more quickly, reducing development costs and resulting in faster and more secure applications.

3.2 Design workflow

To create this framework, multiple aspects must be taken into consideration. Figure 3.1 shows a graphical representation of the required workflow.

3.2.1 Application selection

Predicting the performance for different kinds of functions necessitates training machine learning models with various realistic data. Ideally, utilizing many different functions from real applications would be the preferred way to obtain this. There are, however, some limitations that must be taken into account:

- Code access: the source code of the application must be accessible since the Tigress obfuscator works at the source level;

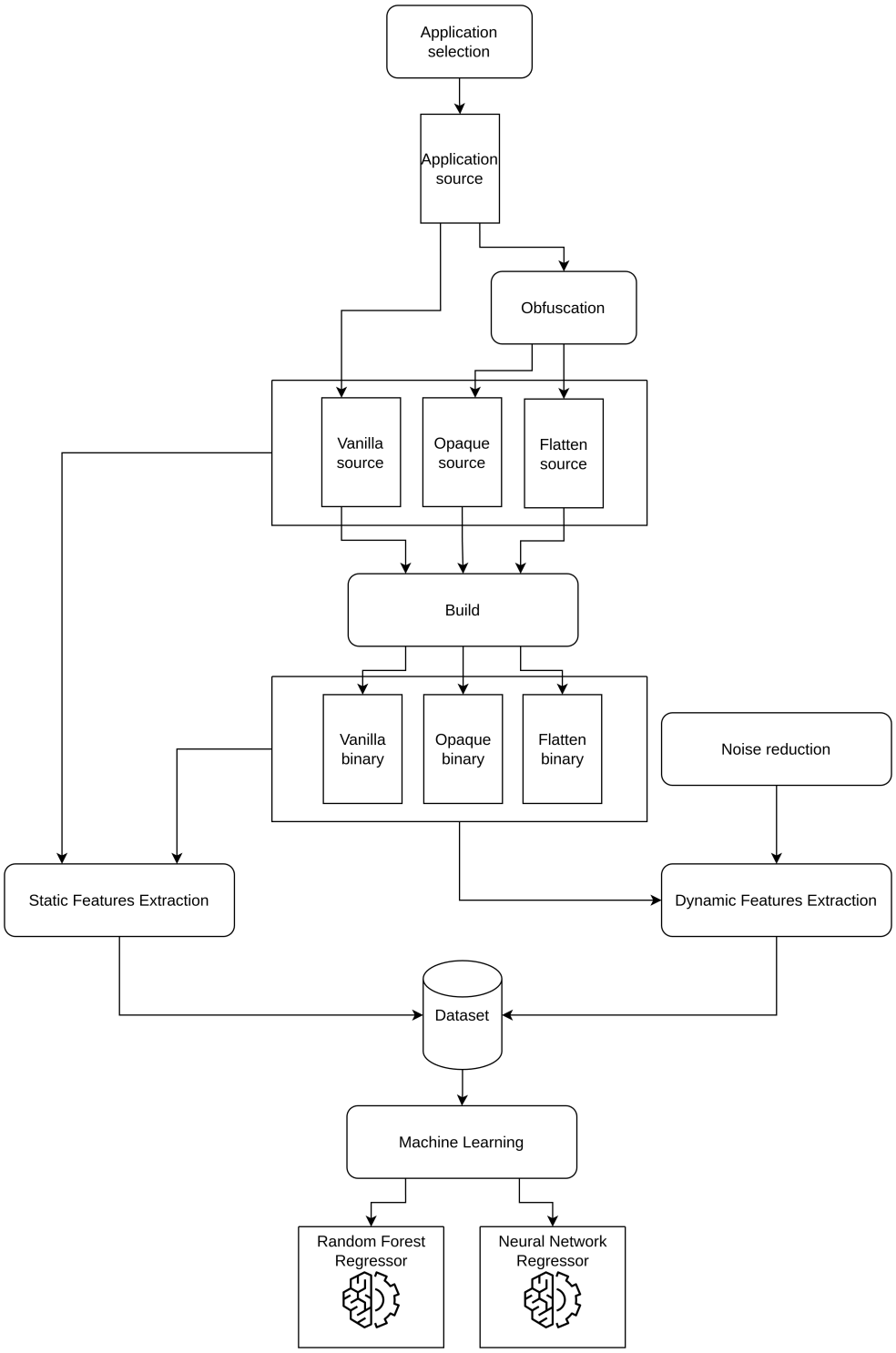


Figure 3.1: Design workflow

- Compatibility: the Tigress obfuscator only works on the C99 version of the C language, limiting the application pool to applications of this type;

- Consistent results: since measuring performance metrics is a very complex task, limiting indeterminism as much as possible can help obtain more consistent results. For this reason, multi-threaded applications and applications where the execution path depends on external factors different from the input (random values, time, ...) are avoided;

- Complexity: applications with too simple functions are avoided. This is related to the fact that simple obfuscation techniques won't alter their execution too much. This will result in almost indistinguishable data between the original and the obfuscated version, making both the obfuscation and measurement processes useless.

3.2.2 Obfuscation

Many different obfuscation techniques exist, and they can be layered on top of each other in different orders, creating a significant number of combinations [10]. Since the obfuscator used, Tigress, works at the source level, only modifications applied to the source code are considered, but binary techniques also exist. The final framework would need to include many more techniques so that developers have more options to choose from. As a starting point, two simple techniques are used to verify the goodness of the results: opaque predicates and code flattening.

- Opaque predicates
  This technique consists of adding to the code expressions with known evaluated values, which still need to be computed at runtime to make the reverse engineering of the application's control flow harder. A simplified representation of this technique can be seen in Figure 3.2.

- Code flattening
  This technique restructures the control flow of an application by aligning each basic block at the same level as the others, as seen in figure 3.3. In this way, if an attacker obtains the application's control flow, it will not be obvious to understand the actual flow of the program.

The selected techniques are applied to the source code of each program. This will produce an obfuscated source code for each technique used.

3.2.3 Build

At this point, three source code versions are generated for each application: the original (vanilla) and the obfuscated versions (opaque and flattened). These are then compiled into their corresponding executable binary code.

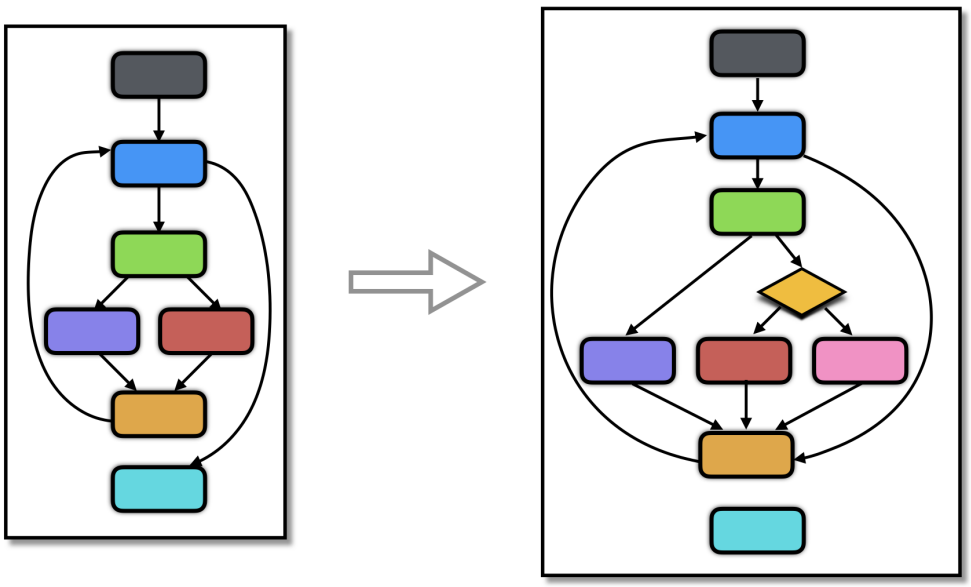


Figure 3.2: Opaque predicates transformation, from `www.tigress.wtf`

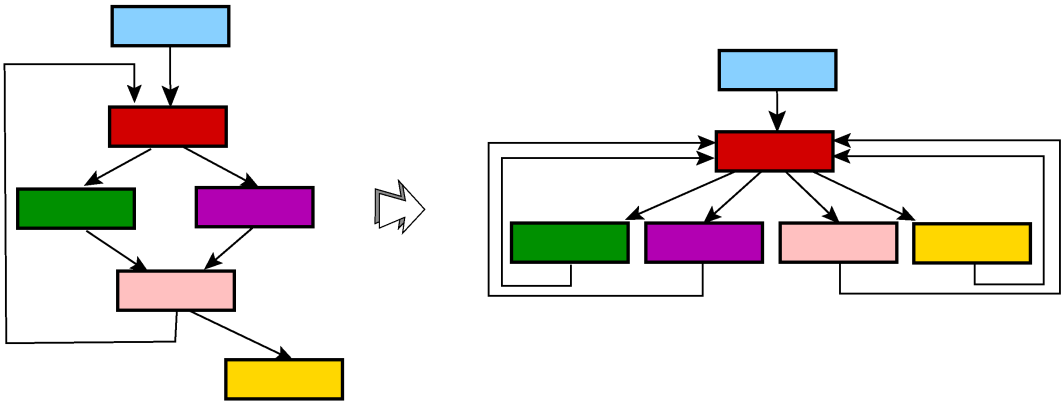


Figure 3.3: Code flattening transformation, from `www.tigress.wtf`

3.2.4 Features extraction

Choosing the best features to characterize applications in the dataset remains an open question. It is still not known which features correlate most effectively with performance. The previous strategy used the frequency of opcodes bigrams: all opcodes, the basic building

blocks of assembly instructions, were recorded for each function executed. The resulting dataset's rows represented a single function execution, and each column represented a pair of consequent opcodes. The value of a cell represented how many times that pair of opcodes was present in executing the selected function. We decided to use a different approach, extracting metrics correlated with the application source code and hardware interactions.

Static features extraction

After the build phase, information can be extracted from the different versions of the application, in both source and binary format, to find differences. Some of these features will impact the machine learning process more, leading to more accurate predictions. In this phase, metrics that can be computed by analyzing the source and binary code statically are extracted without the need to execute the application.

Noise reduction

Modern operating systems and CPUs use many advanced techniques to achieve better performances and lower power consumption. While these features significantly improve the end-user experience, this comes at the expense of the reproducibility of results. Some of these techniques, such as the ones related to CPU frequency scaling, are also non-deterministic, since they depend on the temperature reached by the cores. External factors must be minimized to obtain performance results that closely reflect the application code. Multiple precautions are taken to create a controlled measurement environment to execute our application.

Dynamic features extraction

There are multiple ways to define and measure the performance of an application and its functions. Some common metrics that can be used are the time elapsed, the CPU cycles elapsed and the number of instructions executed.

In the previous thesis, the strategy to measure the performance was to use code instrumentation to query the Time Stamp Counter to get the cycles elapsed during the execution of each function. The process was responsible for calling the instruction to query the counter during execution. This means the measurement is performed in user space since the process does not have kernel privileges.

This approach was evolved to be able to extract more metrics related to performance. To do this, Performance Monitoring Counters (PMCs) are leveraged to extract additional information. They can provide both the common metrics and additional low-level counters that can help understand the application performance behavior better. These features are typically used in workload characterization and include metrics only known at the Operating System level or the CPU level, such as the number of context switches that occurred during execution or the number of branch mispredictions. Hundreds of different events exist for each CPU architecture, but emphasis was placed only on the most commonly

utilized events of the Intel architecture. Metrics from 36 events related to different aspects of the execution pipeline were collected from the PMCs: CPU cycles, branch predictions, context switching, cache hits and misses and others.

`perf` [11] is one of the most commonly used performance analysis tools for accessing PMCs. It has been included in the Linux kernel since 2009 and can be used via the command line. It offers two strategies to access PMCs values: counting and sampling modes. In counting mode, the total number of events of each type that occurred during execution can be read. This is done by resetting the counters' values at the start of execution and checking their values when it ends. In sampling mode, the interrupts are placed at regular intervals every N clock cycles, during the entire program execution. When the interrupt is called, the Instruction Pointer, the call stack and potentially other program state information are saved. This way, when the program ends, it is easy to see which functions were executed more often. This measurement can be faster but less precise since we don't have any information about what happens between two consequent interrupts. Both methods were explored, but since getting accurate data was the priority, only the results from the counting mode were used to create the dataset.

The final dataset will contain all features related to the three versions of the application.

3.2.5 Machine Learning

The ultimate goal is to obtain a statistical model that predicts what performance the obfuscated version of an application would have, only given its vanilla metrics. Machine Learning offers many methods to build a regressor, which is a predictor that takes the features from our dataset as input, and outputs an estimation of the value of one target variable. This target variable can be any variables closely related to performance, such as CPU clocks, time elapses, or the number of instructions.

Two of the most common machine learning techniques considered are:

- Random forest regressor
  A decision tree is a tree in which each node contains a splitting choice based on one of the features. The splitting continues until the leaves contain the estimated value of the target variable. The Random Forest regressor is an evolution of this technique that generates multiple decision trees on subsets of the dataset, and outputs the average of their predictions. This leads to higher accuracy and resistance to outliers. Another advantage is that it does not require extensive hyperparameters tuning, which simplifies the training process. Their output is also explainable, which is an added benefit that can lead to a better understanding of both the model, and the reasons behind different predictions for different techniques. A graphical visualization of this technique can be seen in Figure 3.4. This technique was also used in the previous framework, generating different datasets.

- Neural network regressor
  Neural Networks operate by linking weighted nodes, called neurons, through weighted edges and sending real numbers called signals throughout the network. These weights

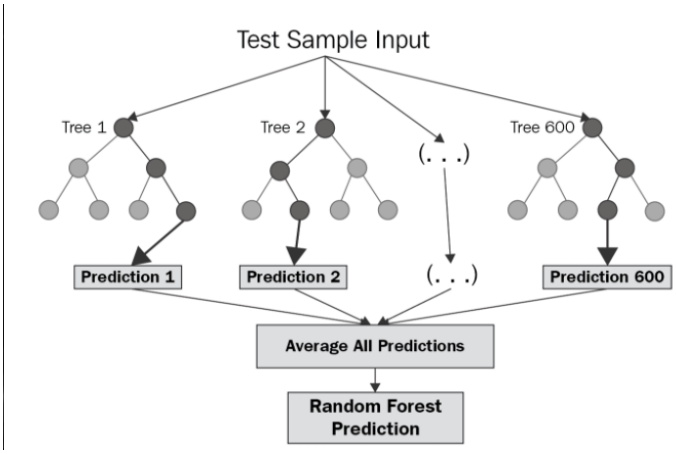


Figure 3.4: Random forest regressor [12]

change as learning progresses, influencing the signal's strength along each edge. The output of each neuron is determined by an activation function, which is a non-linear function of the sum of its inputs. This approach was chosen since the problem of predicting performance is not well studied in the literature and trying a completely different approach may have led to better results. A simplified neural network representation can be seen in Figure 3.5.

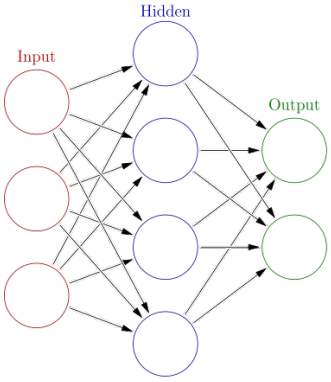


Figure 3.5: Neural network regressor [13]

3.2.6 Model Evaluation

Several metrics [14] were selected to evaluate the performance of both Random Forest Regressors and Neural Network Regressors. These are computed by comparing the actual values of the validation set target metric, with the values predicted by the model.

Mean Squared Error (MSE)

Measures the average squared difference between the predicted values ($\hat{y}_i$) and the actual values (y_i)

$$\text{MSE} = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

- n: number of samples
- y_i: actual value for sample i
- $\hat{y}_i$: predicted value for sample i

Root Mean Squared Error (RMSE)

Represents the standard deviation of the errors (difference between predicted and actual values). It's calculated by taking the square root of the MSE.

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}$$

Standard Deviation

Standard deviation measures the samples spread out from their mean value in the actual values array. A lower standard deviation indicates that the samples tend to be very close to the mean. A higher standard deviation indicates that the samples are spread out over a larger range of values.

This value can be used as a benchmark as it corresponds to the RMSE of a regressor that always returns the average value of the samples metric. A model that achieves a RMSE inferior to this, is capturing some meaningful patterns in the data and making predictions closer to the actual values than just using the average.

$$\textit{Standard Deviation} = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \overline{y})^2}{n}}$$

- n: number of samples
- y_i: actual value for sample i
- $\overline{y}$: mean of the actual values

Mean Absolute Error (MAE)

Measures the average of the absolute differences between the predicted values ($\hat{y}_i$) and the actual values (y_i) for all n samples.

$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}$$

Mean Absolute Percentage Error (MAPE)

Measures the average of the absolute percentage errors. For each sample, it calculates the absolute difference between the predicted and actual values, divides it by the exact value (y_i), averages this percentage error over all n samples and multiplies by 100 (to express as a percentage).

$$\text{MAPE} = \frac{\sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right|}{n} \times 100$$

R-squared (R^2)

R-squared is a statistical metric used to assess how well a regression model explains the variance in the dependent variable (y_i) based on the independent variables included in the model. It essentially tells how much of the variability in the actual data can be explained by the model's predictions.

A low R-squared value (<= 0) suggests that the model fails to capture any of the variance in the dependent variable around its mean. A higher R-squared value indicates a better fit of the regression model to the data. An R-squared value of 1 indicates that the model perfectly accounts for all the variability in the dependent variable around its mean.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2}$$

- n: number of samples
- y_i: actual value for sample i
- $\hat{y}_i$: predicted value for sample i
- $\overline{y}$: mean of the expected values

Percentile Metrics

Values at specific percentiles of the actual data distribution. The percentile is computed by determining the proportion of values that fall below a particular value, relative to the total number of values, then multiplying by 100 to express it as a percentage.

Prediction Error Rate

The absolute error between predicted and actual values was calculated for each validation sample to understand the implications of the model performance better. Samples were

then grouped according to their error magnitude. This analysis reveals the distribution of errors, allowing evaluation of how many predictions fall within a certain percentage of the actual values.

Chapter 4

Implementation

The following chapter explains how each step of the project described in the Design was implemented.

4.1 Application Selection

In comparison with the previous thesis, we only kept csv2latex and dadadodo, while adding 13 new applications: apl, benchmark-io, bzip2, fastdnaml, huffman, lua, mbw, nrg2iso, sha256_rfc6234, smaz, stegx, wav2c, wavheader.

Adding these files resulted in a pool of 74990 lines of code distributed across 528 unique functions. Table 4.1 shows all the applications used with their codebase size.

Choosing the configurations and inputs for each application is not easy.

To guarantee a wide range of input data, the previous strategy was to select some applications and then use a specialized program designed to generate different application inputs to discover as many different execution paths as possible. This approach, called fuzzing, was found to be very time-consuming. This also limited the potential applications pool to applications with short execution time, since many executions were required to find new paths.

In theory, this approach could provide many different workloads for the same application, increasing the diversity of the dataset without requiring new applications. In practice, without an additional effort on the fuzzer configuration, the generated paths are at risk of being very short and similar between them. This would result in wasted time for the fuzzer activity and a resulting dataset with many repetitions of similar data, which would not help the machine learning model generalize the result to different conditions. Moreover, having a larger dataset size would require more training time, without necessarily leading to the significant improvements one might expect from such an effort.

We noticed this was the case for some of those paths generated and decided to take a different approach. We relied on the example inputs given by the authors of each program and only used them, since they would represent meaningful execution paths. We also

Name	Description	LOC
apl	A programming language known for its unique array-oriented syntax and mathematical functions.	17719
benchmark-io	A tool used to measure and compare the performance of different I/O operations.	548
bzip2	A file archiver that uses the Burrows-Wheeler transform and Huffman coding for data compression.	8099
csv2latex	A tool that converts comma-separated values (CSV) files into LaTeX tables, a format commonly used for typesetting tables in scientific documents.	689
dadadodo:	A tool that analyses texts for Markov chains of word probabilities	6601
fastdnaml	A software library for performing fast DNA sequence analysis.	4685
huffman	A program that implements Huffman coding, a technique for lossless data compression.	333
lua	A lightweight scripting language often used for extending applications or adding functionality to websites.	17077
mbw	A software package designed for manipulating and analyzing biological sequences.	304
nrg2iso	A tool used to convert NRG (Nero Burning ROM) image files to the ISO disc image format.	148
sha256_rfc6234	A program that calculates the SHA-256 hash of a file, following the specifications of RFC 6234.	932
smaz	A file compressor utilizing a dictionary-based approach for data compression.	278
stegx	A steganography tool used to hide data within image files.	16670
wav2c	A program that converts WAV audio files to a compressed format, potentially referring to various formats like MP3 or AAC.	378
wavheader	A tool used to manipulate or analyze the header information of WAV audio files.	529

Table 4.1: List of applications used

selected some standalone benchmark-like applications, like apl and benchmark-io, where no input was required. This guarantees that, even if each application only follows a few execution paths, each of them brings diversity instead of redundancy in the dataset.

4.2 Obfuscation

Each application was obfuscated using the Control Flow Flattening and Opaque Predicates obfuscation techniques. Since the priority was improving the framework rather than

expanding the range of obfuscation techniques tested, the same compilation and obfuscation phases were maintained.

The obfuscation process is the same as the previous thesis, and it is composed of different steps:

4.2.1 Prerequisites

Merge source files

If an application is composed of multiple .c and .h files, the `tigress-merge` utility is used to merge all the files into a single one because Tigress only works with a single file.

Add source code modifications

The source code of the applications must be modified by including the header file `tigress.h` and by calling the `init_tigress` function. Tigress authors suggest adding this auxiliary function where Init-transformations can put initialization code. Tigress works by applying different transformations to the functions that are passed as parameters. The Init-transformations are preparatory steps required by specific obfuscation techniques to function properly. A value known by the programmer but still computed at runtime is needed to create opaque predicates. This is called an invariant. Using the `InitOpaque` transformation, Tigress adds data structures with predetermined invariants to the `init_tigress` function. We rely on linked lists and arrays, but different data structures can also be used. In this case, the invariant will be a condition or a property that always holds true in that data structure.

Extract functions names

Tigress expects the names of all the functions to be obfuscated, so we automatically extract them from the merged source file using the `ctags` tool. This common tool can be used to scan files looking for various language objects and generate a "tag" file. This file serves as an index that associates each symbol name with its corresponding information:

- type of symbol (functions, variables, macros, classes,...)
- file name containing the symbol
- line number where the symbol appears
- source line where the symbol appears

By parsing the tool output, function names for each application are retrieved.

4.2.2 Obfuscation Transformations

To understand the inner logic of an application, it is often helpful to have a graphical representation of the code that helps highlight the connections between the different sections. A Control Flow Graph represents in a graph all the possible execution paths of the application, simplifying the reverse engineering process. The compiler also uses this to perform multiple types of optimizations, like eliminating unreachable code, rescheduling instructions order or improving register allocation to variables.

The `gcc` compiler provides the `-fdump-tree-cfg-graph` option [15] that generates a `.dot` file describing the Control Flow Graph generated during compilation. We can generate an image of this file using the `graphviz` tool. We will use the `sha256_rfc6234` application to show this:

```
gcc -fdump-tree-cfg-graph vanilla.source.c -o vanilla_cfg.exe
dot -T png vanilla.source.c.012t.cfg.dot -o vanilla_cfg.png
```

The image generated in this way is huge, since it includes the control flow graph for all the functions in the application.

Luckily, the `dot` file includes one `subgraph` for each function, so it is easy to modify it to include only the desired functions.

We will use as an example the `SHA224_256PadMessage` function, which is a non-trivial function of reasonable length, used in a real implementation of the `SHA256` algorithm.

From Figure 4.1, it is possible to see that it includes two loops and 12 basic blocks.

A Basic Block is a sequence of instructions with a single entry and exit points. This means that among its instructions, only the first one will be reachable from other program instructions, and only the last one will jump to a different basic block. Dividing groups of instructions into basic blocks simplifies the compiler's analysis of the execution flow.

Opaque

After the `InitOpaque` transformation prepares the necessary data structures, the `AddOpaque` transformation is executed. It splits up the control flow of the application and adds opaque predicates. This can be customizable using different parameters:

- Count: How many opaques to add to each function. We decided to use 10.
- Kinds: the branch where the statement should be executed. Which types of bogus computation should be inserted in the newly created fake branches? The following options were selected:
  - `true`: executes the Real Statement in the correct branch
  - `call`: in the fake branch calls a random existing function

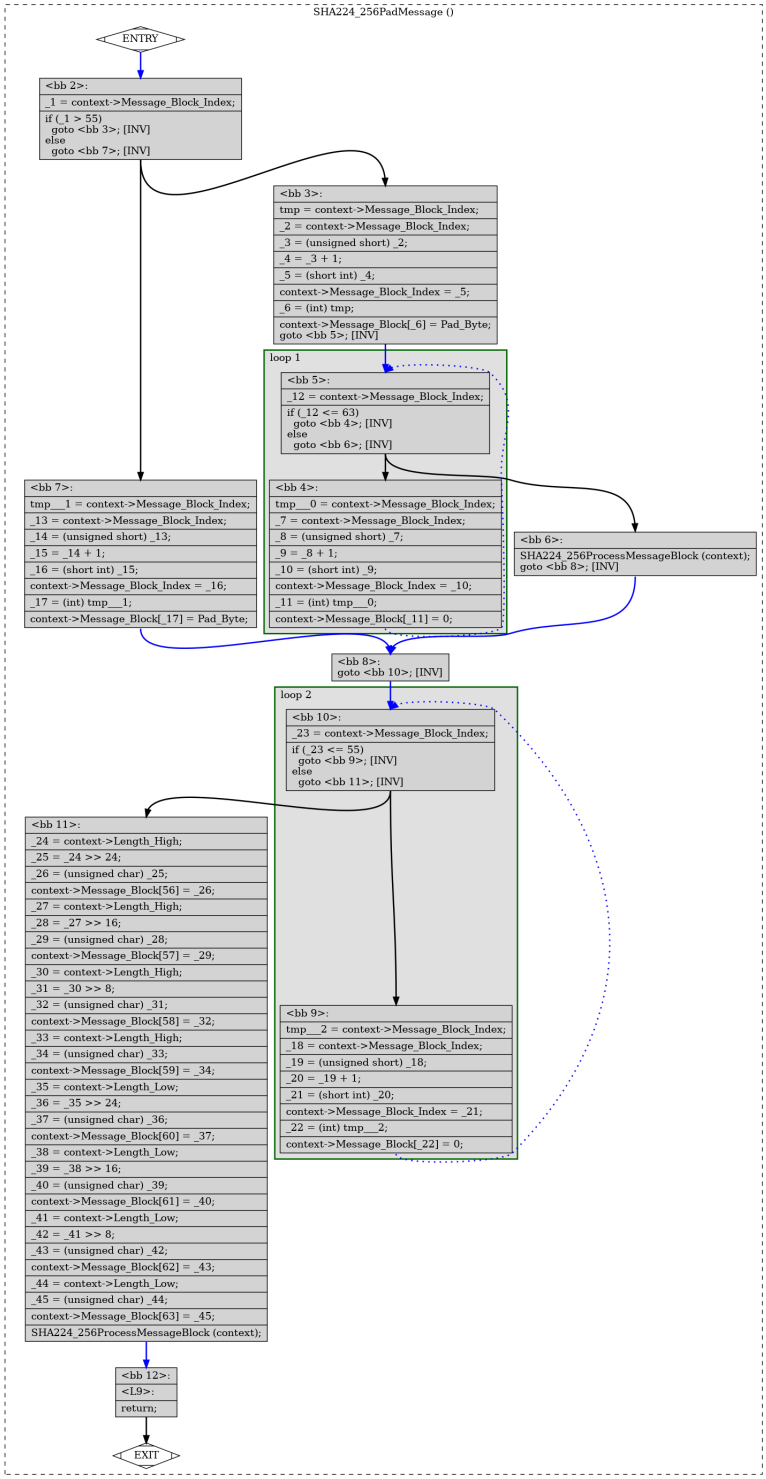


Figure 4.1: Control Flow Graph of SHA224_256PadMessage vanilla function

- `bug`: in the fake branch executes a mutated version of the Real Statement, with bugs inserted
- `junk`: in the fake branch executes fake assembly code like `asm(".byte random bytes")`

From Figure 4.2, it is possible to see that the opaque function still includes two loops, but it now has 29 basic blocks and ten new if-statements that use opaque predicates. Those can be easily identified from the variable names like `1_init_tigresss_1_opaque_ptr`. This graph is computed starting from the obfuscated source. At the same time, an attacker would probably deal with the graph generated from the decompiled version of the obfuscated executable, which would not have the variable name information. It is clear to see that the effort needed to reverse engineer this function is higher than the vanilla version.

Flatten

This transformation does not require additional init-Transformations so that it can be directly applied to the merged source code. The goal of this transformation is to hide the normal control flow of the application by putting all the basic blocks on the same level and using a new variable to keep track of the next block that has to be executed.

- Dispatch: Method used for the dispatch block. We use as dispatch block a `switch` statement wrapped inside an infinite loop. Each basic block becomes one of the switch cases and the value of the switch variable controls which block to execute next.
- ObfuscateNext: Decides if the dispatch variable has to be obfuscated with opaque expressions or not. We do not obfuscate it.
- ConditionalKinds: How to transform conditional branches. We use normal branches with `goto` and the corresponding label.

The resulting Control Flow Graph (Figure 4.3) only has one loop, which incorporates the original loops and most other basic blocks. As a result, the total number of basic blocks increased from 12 to 24. Also, in this case, compared with the vanilla version, reconstructing the original control flow is not straightforward and may require executing the application.

4.3 Build

The vanilla and obfuscated source of each application need to be compiled into the corresponding executable. Each application can specify its compilation rules in a Makefile to simplify the building process for the user. With it, the user can simply use the `make` command to automatically compile the application. Using `make -n` it is possible

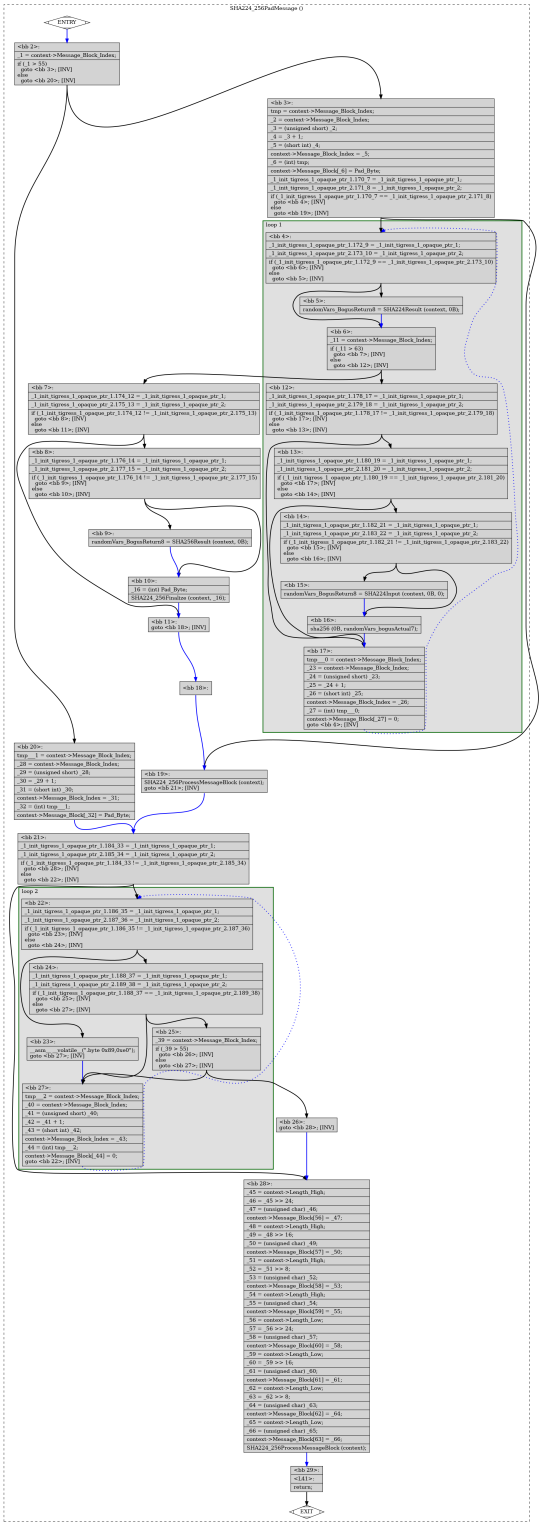


Figure 4.2: Control Flow Graph of SHA224_256PadMessage opaque function

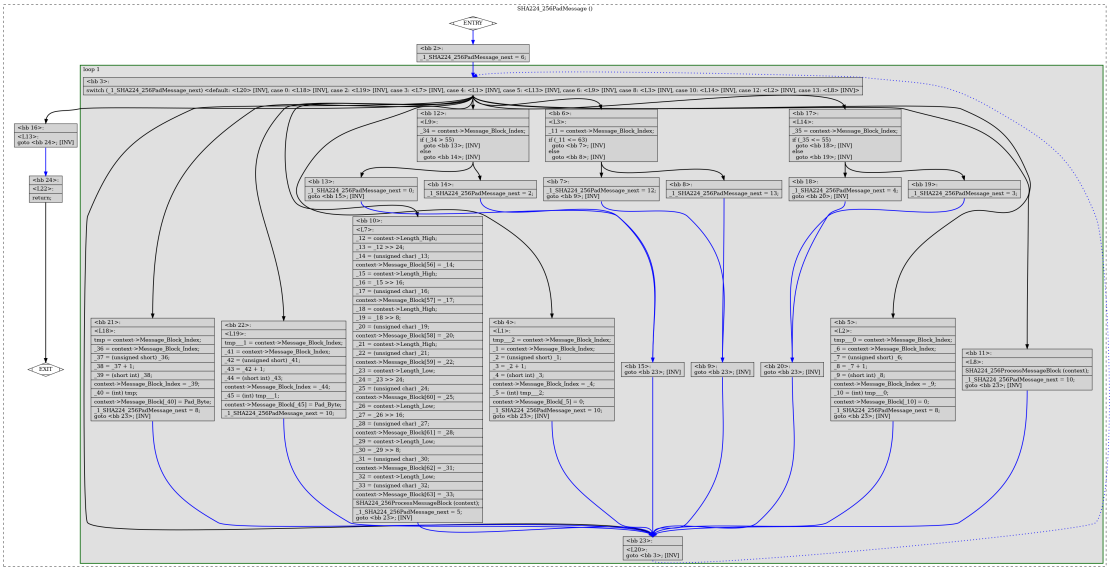


Figure 4.3: Control Flow Graph of SHA224_256PadMessage flatten function

to see the final command generated by the `make` tool after following the compilation rules. This command or `make` will lead to the same result. In order to keep track of the applications used in the framework, the `compiling_dict.json` file was created. It includes various information about the application, like the GitHub link if available, a comment explaining the application, its build command and its argument. In addition to the compiler flags indicated by each application developer, some additional flags were added in every compilation:

- `-g`: this option instructs the compiler to keep debugging information as line numbers, symbols names, or other information required to reconstruct the call stack. This information is useful for the measurement process

- `-fno-omit-frame-pointer`: with this option the compiler will keep the frame pointer instead of optimizing it away. This is another crucial information used to reconstruct the call stack since it is the pointer to the start of the current stack frame.

- `-fno-inline-functions`: this is another option that avoids a compiler optimization, which is inlining. This technique consists of copying the code of a function directly on the call site to avoid the function call overhead. This must be avoided since we need to compare function executions across the different application versions, so we need to be sure that function calls are not optimized away.

4.4 Static features extraction

For the machine learning model to provide accurate predictions, it must have a way to recognize similar functions, so different types of function metrics are needed. Many static features can be combined according to various criteria to estimate the functions' complexity and categorize them. These metrics are extracted from the vanilla source code and the assembly code obtained after disassembling the executable code.

4.4.1 Source code metrics

The following metrics are extracted from the C code using the Multimetric [16] Python tool, customized to work with specific functions instead of the whole file.

- Cyclomatic Complexity: Measures the complexity of the function's control flow graph as a function of the number of edges, nodes and connected components in the graph [17]
- Fanout: Considers function dependencies on other parts of the program (fanout internal) and on third-party libraries or system calls (fanout external)
- Lines of Code: Provides a basic indication of code size after cleaning and reformatting it
- Operands: Represent how many variables or constants are being manipulated. Both the total number and the unique number are extracted
- Operators: Represent how many operations are performed on operands. Also, in this case, both total and unique numbers are extracted
- Halstead Metrics: Estimate software complexity as functions of the total and unique number of operands and operators present [18]
  - Volume: Indication of the program's overall size
  - Difficulty Level: Indication of the perceived mental workload required by the developer to understand the code
  - Effort: Indication of the effort required to develop and maintain the code
  - Time: Indication of the time required to develop and maintain the code
  - Bug: Indication of the number of defects that could be present in the code
- Tiobe Quality Score: Combines using different weights various metrics like cyclomatic complexity, fanout, code duplication, adherence to standards and security practices into a single score reflecting overall code quality [19]

4.4.2 Assembly metrics

The following metrics are extracted from the vanilla executable using the Radare2 tool [20]. Among all the information that it can extract, like function signatures, local variables and stored strings, there are also two useful performance metrics:

- Cyclomatic Complexity: Computed on the assembly control flow graph, can reveal how compiler optimizations affected the code complexity
- Cycles cost: Estimated number of cycles that would be taken during the function execution

By using the `objdump` tool, each function was disassembled and their instructions were grouped into different categories (Memory, Stack, Arithmetic, Logic, Jump, Shift, Call, SysCall) with the help of regexes. Having the frequency of each type of instruction for each function may provide insights that the machine learning model could use to find similarities between different functions, which could lead to similar performance behavior after being obfuscated in the same way.

4.5 Noise reduction

In a perfect world, it would be possible to have a single number that represents the performance of an application function. In reality, this cannot happen because such a number does not exist. Different inputs can lead to different execution paths with different instructions, resulting in a different performance. This can easily be taken into account by considering the same function with different inputs as different entities.

The main problem resides in the fact that each function run acts in a different environment, where hardware features and operating system features can interfere with the execution, altering its performance. It is unsurprising to expect different performances from the same function to be executed on different hardware or operating system versions. What is less obvious is that in modern systems, it is almost impossible to obtain identical measurement values, even when running it twice on the same machine.

Since our goal would be to obtain as close of a performance value as possible for each version of our application so that we could find how to generalize these values to other applications, we need to find ways to minimize the effect of the various interferences.

Note that even if some of these techniques may alter the "value" of the application performance compared with the performance of the application executed in a "real" environment, the result will still be significant, since the "value" that we find is more independent from external factors and can be generalized to "real" environment, while the contrary is not true. If we did not use these techniques to limit noise and just measured performance in a "real" environment, it would be much harder to generalize them to different environments where the noise impact is different, while in this way, the performance result will be correlated to the application itself.

The `benchmark_system_config.py` Python script was created by properly configuring hardware and operating system features to obtain a stable and low-noise environment where consistent performance measurements could be achieved.

4.5.1 Support functions

Since many techniques used to reduce noise involve setting system variables by writing on system files or executing privileged operations, two support functions were created to simplify development.

```
def set_system_variable(filename, new_content):
    previous_content = run_command(f"cat {filename}")
    run_command(f"echo {new_content} > {filename}", sudo=True)
    logger.debug(f"{previous_content} -> {new_content}")
    return previous_content

def run_command(command, sudo=False):
    if sudo:
        command = f'sudo sh -c "{command}"'
    result = subprocess.run(
        command, shell=True, capture_output=True, text=True, check=True
    )
    return result.stdout.strip()
```

4.5.2 Hardware and software consistency

The most straightforward way to keep performance measurements consistent is to take all of them on the same machine without changing Operating System or Software versions across the measurement. For this goal, a function was created that checked the versions of the Operating System, Kernel, GLIB, and Compiler used, and stopped execution in case those were different from the ones initially set.

```
def check_version(actual_version, expected_version, message):
    if actual_version != expected_version:
        logger.debug(f"Wrong {message} version: {actual_version} != {
    expected_version}")
        exit(-1)

def check_versions():
    OSVersion = run_command(
        "cat /etc/os-release | grep 'PRETTY_NAME'| cut -d= -f2 | tr -d
    '\"'"
    )
    check_version(OSVersion, "Ubuntu 21.04", "OS")

    KernelVersion = run_command("uname -r")
```

```
13      check_version(KernelVersion, "5.8.0-59-generic", "kernel")
14
15      GLIBVersion = run_command("ldd --version | head -n 1")
16      check_version(
17          GLIBVersion, "ldd (Ubuntu GLIBC 2.33-0ubuntu5) 2.33", "GLIBC
    library"
18      )
19
20      CompilerVersion = run_command("gcc --version | head -n 1")
21      check_version(
22          CompilerVersion, "gcc (Ubuntu 10.3.0-1ubuntu1) 10.3.0", "GCC
    compiler"
23      )
```

4.5.3 Dynamic Frequency Scaling and Scaling Governor

The number of instructions that a processor can execute in a unit of time is affected by a large number of factors, including its clock frequency. For this reason, a fixed frequency will lead to more consistent results than a variable frequency. For this to happen, the default configuration for the Dynamic Frequency Scaling feature needs to be changed, disabling it.

This feature, called Intel Turbo Boost for Intel processors, is typically enabled because it allows the processor to adapt its frequency depending on the computation needs of the moment. This lets the processor keep a lower frequency when idle or during low-load operations, resulting in lower power usage and heat generation. At the same time, when the operating system requests the highest performance state of the processor, with this technology, the frequency of the processor can increase to meet the new demand. While this process results in a better overall user experience, with lower consumption or higher performance depending on the need, this increases variability and indeterminism when evaluating application performance. This is because the operating system decides which power state to request after evaluating multiple metrics, like CPU load and temperature, that cannot be controlled.

By disabling this feature, the processor's frequency remains constant regardless of the requested load.

The processor's frequency should be kept constant; it is necessary to set the behaviour of the Linux Kernel Scaling Governor to one of the static options. We decided to use the `performance` option, which sets it to the maximum frequency supported by the processor. [21]

```
1
2   logger.debug("Disabling CPU Dynamic Frequency Scaling (Turbo mode
        functionalities)")
3   previous_no_turbo = set_system_variable(
4       "/sys/devices/system/CPU/intel_pstate/no_turbo", "1"
```

```
)

logger.debug("Disabling all Power Optimizations (Linux Scaling Governor)
    ")
previous_scaling_governor_statuses = {}
for cpu_num in [0, 1, 2, 3, 4, 5, 6, 7]:
    previous_scaling_governor = set_system_variable(
        f"/sys/devices/system/CPU/cpu{cpu_num}/cpufreq/scaling_governor"
    ,
        "performance",
    )
    previous_scaling_governor_statuses[cpu_num] =
    previous_scaling_governor
    logger.debug(f"Cpu {cpu_num} in performance mode")
```

4.5.4 Simultaneous Multithreading

Modern superscalar CPUs use the Simultaneous Multithreading technique to improve the overall efficiency of each physical core. This allows multiple (typically 2) completely different tasks to be executed on the same physical core simultaneously. This means they share access to the execution pipeline, the ALUs, the caches and other physical resources. Since the various resources will have less idle time, this allows for higher overall throughput and power efficiency of the core. On the other end, each single task may run slower and will have less predictable performance since it will depend on the other task that was scheduled to run on that core and on the resource competition.

This is implemented on Intel processors to show double the number of physical cores available as CPUs. For example, in the Intel i7-4910MQ machine used, this behavior can be seen using the `lscpu` tool:

```
> lscpu
...
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
...
```

This means that even if only four physical cores are actually present, eight possible threads can be executed simultaneously, since Simultaneous Multithreading is active. To disable this behavior we need to find which of those CPUs are the physical ones, and which are the "virtual siblings" CPUs, and disable the "virtual" ones.

```
> cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
0-1
> cat /sys/devices/system/CPU/cpu2/topology/thread_siblings_list
```

```
2-3
> cat /sys/devices/system/CPU/cpu4/topology/thread_siblings_list
4-5
> cat /sys/devices/system/CPU/cpu6/topology/thread_siblings_list
6-7
```

This shows that the physical CPUs are indicated by the numbers 0,2,4 and 6, while the siblings we should disable are 1,3,5,7. Now, we can disable them by writing on the corresponding system file

```

logger.debug(
    "Disabling Simultaneous Multithreading (Intel Hyper-Threading
    technology)"
)
previous_online_statuses = {}
for sibling_thread in [1, 3, 5, 7]:
    previous_status = set_system_variable(
        f"/sys/devices/system/CPU/cpu{sibling_thread}/online", "0"
    )
    previous_online_statuses[sibling_thread] = previous_status
    logger.debug(f"Thread {sibling_thread} disabled")
```

We can see that the effect was achieved by repeating the `lscpu` command:

```
>lscpu
...
CPU(s):                          8
On-line CPU(s) list:             0,2,4,6
Off-line CPU(s) list:            1,3,5,7
Thread(s) per core:              1
...
```

4.5.5 CPU affinity

Modern superscalar CPUs leverage the multiple cores present on the chip to handle multiple tasks and applications at the same time. At any given time, many system processes are running to handle various aspects of the PC functionalities. On top of those, the user can run multiple applications simultaneously. All these tasks need to share the same processor resources, and the operating system is in charge of scheduling the various tasks to the various cores at any given moment.

This can become a problem when measuring a single application's performance because it may be affected by the other processes running on the machine at that moment. The

problem arises mainly when the Scheduler performs a context switch or a CPU migration. In both cases, the application execution is stopped, and its state is saved so that it can later be resumed, either on the same core or on a different core. Both of these events slow down the application performance because some system resources, like the pipeline and the caches, will not be saved and restored.

By minimizing the number of context switches and CPU migrations, the performance measurements will be more consistent and will not depend on the overall state of the system as much.

A context switch can be triggered when a different task must be executed or when an interrupt occurs, maybe because of an event caused by the user.

A CPU migration may also be the result of the scheduler trying to load balance tasks from overloaded cores to underutilized ones to increase the overall system performance or trying to concentrate tasks in a single core in order to reduce power consumption by putting other cores in a low-power state.

If we want to minimize both of these events, we can reduce interrupts or reduce the times when a new task could be scheduled on the same core that is running our application.

Disabling interrupts is possible [22]. However, it requires running each application in a custom kernel module where we disable preemption before executing the code we want to measure and reenable it afterwards. This is a big change that would heavily alter the applications and may produce unexpected results due to the missing interrupt feature.

There is another easier solution that leverages the cpuset mechanism to obtain higher control on the execution environment: CPU shielding. Cpusets are groups of cores that can be used to constrain where different tasks will be executed. The root cpuset always exists and includes all the cores on the machine. The `cset-shield` tool allows the creation of the `system` cpuset, which will be reserved for the common background system tasks, and the `user` cpuset, where only user tasks are allowed to run. Using the command in Code 4.5.5, all tasks that are currently running in the `root` cpuset will be moved to the `system` cpuset (cores 0,1,4), including kernel threads (`-k on`). The application and the measurement tool will run in the `user` cpuset (cores 2 and 3). [23]

```
result = run_command("sudo cset shield -c 2,3 -k on")
logger.debug(result)
```

Since context switches and CPU migrations are not entirely avoidable, we still record how many of these happen during measurement so that it is possible to have an idea of the accuracy of the measurement or the degradation of the performance given by the event.

CPU migrations measurements show that the CPU shielding technique is particularly effective since, in the final dataset, over 99,9% of the 3 million function executions recorded have 0 CPU migrations.

4.5.6 Address Space Layout Randomization

Unlike the previous techniques, this is a security technique created to prevent the exploitation of memory corruption vulnerabilities. It involves randomly rearranging the position of key address space areas, like the stack, heap, libraries and base of the executable, in order to make the redirection of code execution more difficult. While it has been seen that this memory layout rearrangement does not have a significant impact on performance, it can still make measurements less consistent or generate outliers, so we decided to disable it.

```
logger.debug("Disabling ASLR")
previous_randomize_va_space = set_system_variable(
    "/proc/sys/kernel/randomize_va_space", "0"
)
```

4.6 Dynamic features extraction

4.6.1 Events

Among the various hardware features that can be used to obtain performance metrics, Performance Monitoring Counters are the most flexible and accurate. They are special-purpose registers present on modern processors in a limited number. For example, our machine's Intel Haswell Architecture processor contains eight such registers. These can be configured to act as counters for any of the hundreds of architecture-specific events available. Each event can provide very specific insights regarding the low-level interactions between software and hardware. The 36 events that were considered are shown in Table C.1. [24]

The table includes the event name used by Perforator (the tool used for Events Sampling), the corresponding event name used by `perf record` (the command used for Events Sampling), the overall event type, a description, and the number of the event group that includes it.

Event Types

- Hardware: This indicates one of the kernel's "generalized" hardware events.
- Software: This indicates one of the software-defined events provided by the kernel.
- Cache: This indicates a hardware cache event

Cache events hierarchy

The various cache events can be categorized according to different factors:

- The physical component considered
  - Branch Prediction Unit: Component that attempts to predict the outcome of conditional branches to improve instruction fetch efficiency.
  - Data Translation Lookaside Buffer: Caches recently used translations between virtual memory addresses and physical memory addresses to speed up data access.
  - Instruction Translation Lookaside Buffer: Caches translations for instruction memory addresses, accelerating instruction fetching.
  - Data Level 1 Cache: High-speed memory that stores frequently accessed data, reducing the need for slower main memory access.
  - Instruction Level 1 Cache: High-speed memory that stores frequently used instructions for faster execution.
  - Last Level Cache: The largest and slowest cache in the CPU hierarchy, acting as a buffer between the main memory and the smaller, faster caches.
  - Local Memory: Main memory (RAM), which stores all programs and data the CPU needs to access.
  - Disk: Non-volatile storage device used for permanent data storage.
- The operation initiated by the CPU when the event was triggered:
  - Read
  - Write
- The outcome of the operation considered:
  - Accesses: considers all attempts to access the component, regardless of the outcome
  - Misses: only considers failed attempts when the requested data/instruction was not present on the component

A simplified representation of the various components can be seen in Figure 4.4

4.6.2 Multiplexing

Since only up to 8 events can be measured at the same time using the available registers, `perf` implements a time multiplexing technique to allow for more events to be considered in the same execution. This consists in granting the use of the same register to different events in consecutive time intervals. At the end of the execution, the final count for an event will be:

$$final\ count = raw\ count \cdot (total\ application\ time / recording\ time)$$

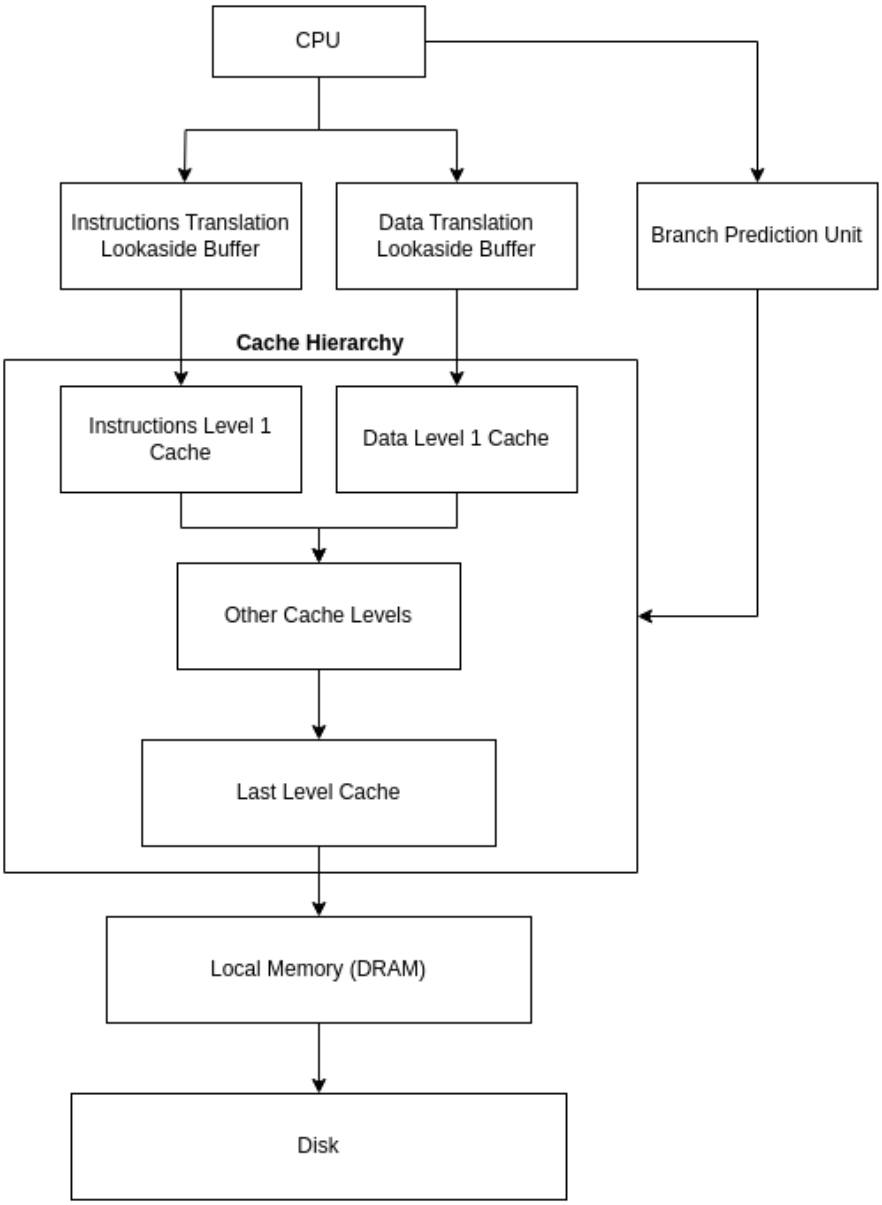


Figure 4.4: Simplified Cache Hierarchy structure

This final count is only an approximation of the actual value, which can be accurate in case of workloads that repeat the same operations over a long period. When measuring applications that run in less time or have variable workloads, the blind spots may significantly reduce accuracy.

Some of the functions in real applications are naturally very short, so the multiplexing technique will not produce meaningful results that could be used effectively by the machine learning model. At the same time, ignoring these functions altogether will decrease the size of the final dataset, making it less promising.

The solution that allows measuring also small functions accurately is to avoid multiplexing, considering only up to 8 events simultaneously. This will produce more accurate results at the expense of needing to run each application one time for each event group.

Events groups

To increase the amount of information about the functions' performance, a large number of events needed to be considered. In order to do so while avoiding multiplexing, events needed to be divided into groups of 8 or fewer events that could be measured simultaneously. The goal was to measure the largest number of events in the fewer executions to obtain more information without spending too much time in the measuring process. This was not an easy task since not all events can be combined to be measured at the same time. Some events have restrictions on which counter they can schedule and may not support multiple instances in a single group. For this reason, a try-and-error phase was performed until a satisfactory grouping was found.

4.6.3 Prerequisite setup

The `perf` tool can be installed through the `linux-tools-generic` and `linux-tools-$(uname -r)` package, which contain the version corresponding to the current kernel. The only requisite is that the Linux Kernel version of the machine must be greater than or equal to 2.6.31.

The primary function that is going to be used is the `perf_event_open` system call. This is a complex function with multiple possible parameters and can be utilized to perform different kinds of analysis. It is possible to verify that the kernel supports it by checking the existence of the `/proc/sys/kernel/perf_event_paranoid` file.

This file also restricts access to the performance counters for security reasons. Depending on its value, the user will be allowed to measure only some or all performance events:

- 2: allows only user-space measurements (default since Linux 4.6)
- 1: allows both kernel and user measurements (default before Linux 4.6)
- 0: allows access to CPU-specific data but not raw tracepoint sample
- -1: no restrictions

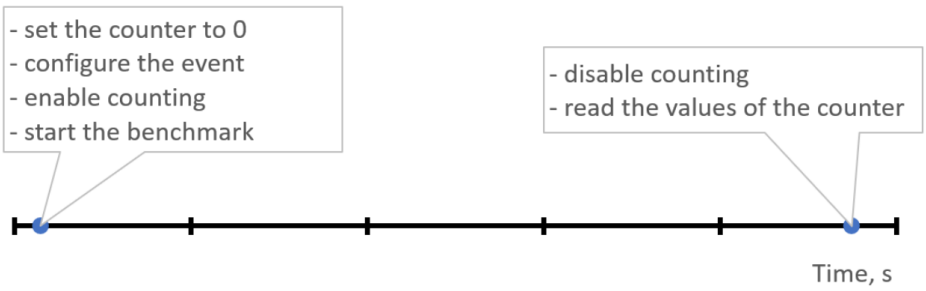


Figure 4.5: Events Counting Mechanism [9]

Setting this to -1 is required for both counting and sampling the types of events that we need.

We simply add this instruction to the `benchmark_system_config.py` script used for noise reduction.

```
logger.debug(
    "Setting perf_event_paranoid to allow use of (almost) all events by
    all users"
)
previous_perf_event_paranoid = set_system_variable(
    "/proc/sys/kernel/perf_event_paranoid", "-1"
)
```

4.6.4 Events counting

The easiest way to utilize the counting mechanism would be to use the `perf-stat` command, specifying the selected application and the events to measure, one group at a time. This works but only provides aggregated values on the whole application execution. Obtaining counter values at the function granularity level is more complex and can be achieved using the `perf_event_open` syscall. This is the underlying function that allows setting up performance monitoring by passing the parameters that will achieve the desired result, like the selected events. It returns a file descriptor that can later be passed to a different system call: using `ioctl`, events can be enabled and disabled; using `read`, the resulting values can be obtained.

The Figure 4.5 summarizes the process.

Perforator

The only missing piece remains how to place the necessary syscalls at each start and end of every function of the program. Luckily, a program called `Perforator` [25] had already

been developed with this exact goal in mind. It puts together different tools to achieve the desired effect. It takes the function that we want to measure and the events of interest as input, and it can produce a CSV file with the result. In order to do so, different steps are required:

1. To find the starting address of the requested function, it uses the `bininfo` Go package, which parses the application information from its ELF binary format
2. It uses the `ptrace` system call to attach itself to the target program and places the `0xCC` "interrupt" instruction at the beginning of the function to be measured. This allows it to regain control when the function is about to be executed
3. At that point, Perforator restores the original code that was temporarily replaced with the interrupt byte, retrieves the return address by reading the top of the stack, and places another interrupt byte at that address
4. Then it will enable the events counting and resume the target process
5. When the function reaches its return address, the next interrupt happens, Perforator disable the events, removes the interrupt, and re-inserts the interrupt instruction back at the start of the function.

The program supports measurements of multiple functions at the same time, but it relies on the same multiplexing mechanism that `perf` uses. For this reason, the only way to guarantee precise results is to measure one function at a time.

`collect_dynamic_metrics`

1. Extract functions names
2. Iterate over each application input, each function and each event group
3. Use Perforator to obtain the desired result:

```
sudo cset shield --exec -- sudo <perforator-application> -V --
    summary --csv --kernel -e <events-in-current-event-group> -r <
    function-to-be-measured> -o <csv-results-file-for-that-
    application-input-function-event_group> -- <application-
    executable> <application-input>
```

The meaning of this command can be explained like this:

- `sudo cset shield --exec --` : Execute the following command in the `user` cpuset created previously in the `benchmark_system_config.py` script.
- `sudo <perforator-application>`: Execute the Perforator program with the following arguments

- `-V`: output verbose debug information
- `--kernel`: include kernel code in measurements
- `-e <events-in-current-event-group>`: list events in the current event group to measure
- `-r <function-to-be-measured>`: specify the name of the function that needs to be measured
- `--csv -o <csv-results-file-for-that-application-input-function-event_group>`: write output in the provided output file in CSV format
- `-- <application-executable> <application-input>`: the actual application to be measured and its arguments

An example of a real call of this command counting the events of the first group (`bus-cycles,cpu-cycles,ref-cycles,instructions,context-switches,cpu-migrations,cpu-clock,page-faults`) obtained during the execution of the `SHA224_256PadMessage` function of the vanilla version of `sha256_rfc6234` with the `input10MB.bin` file as input.

```
sudo cset shield --exec -- sudo ./2023_Obfuscation_Thesis/utils/
    perforator-0.4.1/perforator -V --summary --csv --kernel -e bus-
    cycles,CPU-cycles,ref-cycles,instructions,context-switches,CPU-
    migrations,CPU-clock,page-faults -r 'SHA224_256PadMessage' -o
    ./2023_Obfuscation_Thesis/workdir/results/sha256_rfc6234/vanilla
    .debug/'SHA224_256PadMessage'.event_group_0.measure_1.csv --
    ./2023_Obfuscation_Thesis/workdir/app/sha256_rfc6234/vanilla/
    vanilla_debug.exe ./2023_Obfuscation_Thesis/app/seeds/
    sha256_rfc6234/input10MB.bin
```

Note that there is no way to know for sure the return value of the application since it is executed inside the "shield". From the python `subprocess.run()` function, it is only possible to retrieve the return value of the shielding program. Correctness can be verified from the output of the application, reported in the appropriately generated log files.

4. If the result file from that function takes too much space, it means that the function was executed too many times, so it is probably a very short function. Verifying using the static metrics extracted in the previous phase is possible, but this was not done since only a few functions were affected. Such function is removed from the pool, and it is ignored for the subsequent measurements in order to limit measurement time. The reason is that many measurements of very small functions clutter the dataset, taking much memory without adding new meaningful insights. For all functions, it is useful to have multiple measurements since it can make the measured value more consistent and outlier-resistant, but measuring the same function millions of times becomes redundant and wastes both measurement time and training time later. Moreover, it can be proved that very short functions are impossible to obfuscate, so it is not reasonable to consider them as an actual use case scenario for developers who want to obfuscate their applications.

5. It is possible that the example input provided by the application author leads to an execution path that only covers some application functions. The uncovered function names are saved in a file and excluded from other measurements with the same input. This avoids wasting time executing the application again, trying to measure functions that will not be executed. With new inputs, the function is reinserted into the functions pool. The file containing uncovered functions could also be used to increase function coverage with new inputs.

6. After the application run, the `cleanup_cmd` command specified in the application entry in the `compiling_dict.json` file is executed. This command is used to remove all leftover files from the execution of the application, like the output files generated. This keeps the directory structure clean and avoids potential overwriting errors that could happen when executing the same application again.

7. The CSV files obtained from each function measurement must be concatenated into a single CSV file for each event, like `sha256_rfc6234/vanilla/event_group_0.results.csv`.

4.6.5 Events sampling

Event sampling is one of the most frequently used approaches for performance analysis because it can give enough information in a short time to understand which code region contributes the most to the various performance events. Due to its more straightforward implementation and faster measurement time, this technique was implemented as an alternative to event counting in our framework. As expected, the accuracy of the results was lower than that of the other technique, so they were not included in the final dataset.

Prerequisite setup

Since we can control the frequency of sampling, which means how many samples per second will be collected, we need to ensure that the `perf_event_max_sample_rate` configuration file is consistent with the selected frequency. If we do not, the actual used frequency will be lower, or the command will fail, if the `--strict-freq` was used. Increasing the `perf_cpu_time_max_percent` will instruct the kernel to allow using up to 99% of the CPU time to handle perf event. This could slow down operations, but will guarantee that no samples will be dropped by the kernel in the attempt to reduce CPU usage.

```
logger.debug(
    "Setting perf_cpu_time_max_percent to allow high CPU usage to have more frequent perf events"
)
previous_perf_cpu_time_max_percent = set_system_variable(
    "/proc/sys/kernel/perf_cpu_time_max_percent", "99"
)
logger.debug(
    "Setting perf_event_max_sample_rate to allow more frequent perf events (not required if perf_cpu_time_max_percent=0)"
)
```

```
previous_perf_event_max_sample_rate = set_system_variable(
    "/proc/sys/kernel/perf_event_max_sample_rate", "50000"
)
```

`sample_dynamic_metrics`

1. Extract functions names
2. Iterate over each application input and each event group
3. Use the `perf record` command:

```
sudo perf record -e <events-in-current-event-group> -F 50000 --
    strict-freq -- <application-executable> <application-input>
```

The meaning of this command can be explained like this:

- `sudo cset shield --exec --` : Execute the following command in the `user` cpuset created previously in the `benchmark_system_config.py` script.
- `sudo perf record`: Execute the `perf record` with the following arguments
  - `-e <events-in-current-event-group>`: list events in the current event group to measure
  - `-F 50000`: sample events 50000 times every second
  - `--strict-freq`: if the specified frequency cannot be used abort the operation with an error instead of throttling the sampling frequency
  - `-- <application-executable> <application-input>`: the actual application to be measured and its arguments

This command will trigger a Performance Monitoring Interrupt 50000 times every second. When it happens, a dedicated Interrupt Service Routine will:

- Disable the counter used to trigger the interrupt to avoid ulterior interrupts during the handling of the current one
- Record the current Instruction Pointer of the monitored application to understand what was being executed when the interrupt happened
- Reset and re-enable the counter for the next interrupt
- Resume application execution

A graphical visualization of this technique can be seen in Figure 4.6

At the end of the execution, a `perf.data` file is created, containing information extracted during all previous interrupts.

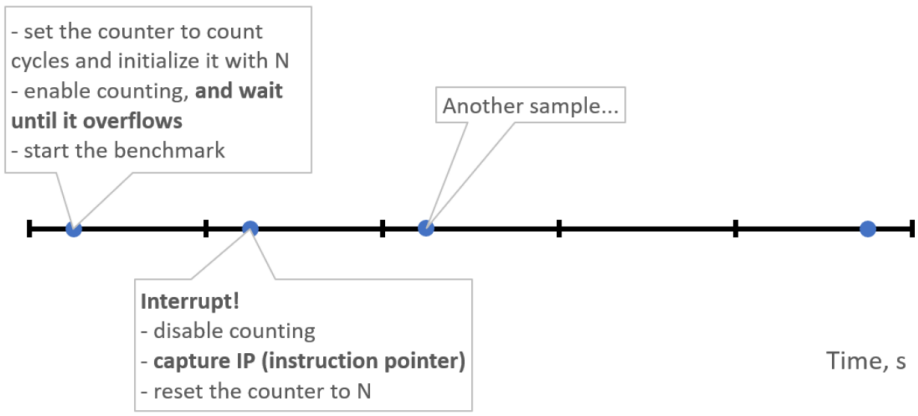


Figure 4.6: Events Sampling Mechanism [9]

An example of a real call of this command sampling the events of the first group (`bus-cycles,cpu-cycles,ref-cycles,instructions,context-switches,cpu-migrations,cpu-clock,page-faults`) 50000 times per second, during the execution of the the vanilla version of `sha256_rfc6234` with the `input10MB.bin` file as input is reported below.

```
sudo cset shield --exec -- sudo perf record -e bus-cycles,CPU-cycles
    ,ref-cycles,instructions,context-switches,CPU-migrations,CPU-
    clock,page-faults -F 50000 --strict-freq -- ./2023
    _Obfuscation_Thesis/workdir/app/sha256_rfc6234/vanilla/
    vanilla_debug.exe ./2023_Obfuscation_Thesis/app/seeds/
    sha256_rfc6234/input10MB.bin
```

4. Use the `perf report` command:

```
sudo perf report --show-total-period --show-nr-samples --fields=
    overhead,sample,period,sym > <report-file>
```

Since the `perf.data` file is in a particular binary format, the `perf report` command is used to visualize the information it contains as needed. In order to obtain a consistent format with the CSV file previously generated by the event counting, data were extracted using these flags:

- `--show-total-period`: Show a column with the sum of periods
- `--show-nr-samples`: Show the number of samples for each of the functions recorded
- `--fields=`: Which fields to display
  - `overhead`: Overhead percentage of sample

- `sample`: Number of sample
- `period`: Raw number of event count of sample
- `sym`: Name of function executed at the time of sample

An example of an actual call of this command generating the report from the `perf.data` previously created.

```
sudo perf report --show-total-period --show-nr-samples --fields=
    overhead,sample,period,sym > ./2023_Obfuscation_Thesis/workdir/
    results/sha256_rfc6234/vanilla.debug/sampling/seed0.
    event_group_0.sample_1.txt
```

5. After the application run, the `cleanup_cmd` command specified in the application entry in the `compiling_dict.json` file is executed. This command is used to remove all leftover files from the execution of the application, like the output files generated. This keeps the directory structure clean and avoids potential overwriting errors that could happen when executing the same application again.

6. `process_reports_into_csv`: create a CSV file for each event group, associating each function to the estimated count and the number of samples of each event

4.7 Machine Learning

4.7.1 Merging results into dataset

All measurement results are contained in CSV files divided per event group measured, like `sha256_rfc6234/vanilla/event_group_0.results.csv`, with one column for each event measured and one row for each function execution recorded. Each value in the table is the value of the counter for that event and that function. Machine learning algorithms use a single dataset as input, so we need to merge all the files into a single CSV file with all functions from all the applications and all events. The merge acts on the `region` and `seed` columns, which represent the function and the application input for that function call.

This is done in two phases: the `csv_merge.py` script generates a single CSV file aggregating all measurements for an application (`sha256_rfc6234.csv`). Even if each event group measurement refers to a different execution of the function, the order of executions is deterministic and repeated for each event group. For this reason, we can merge all the events, even from different groups, into a single row because they are all related to an equivalent function call. Since the `time-elapsed` metric is present in all event group measurements, we decided to associate each function call the average of all the `time-elapsed` since having more measures as different features would be detrimental to the understanding of the importance of the features. Also, all static features related to each function are associated with each call as additional columns.

The second phase consists of concatenating all the application's CSV files in a single file, the final dataset. This is done in the `train_custom.py` script, which also contains the dataset preparation steps, the model training and validation. Having these phases together lets us simplify the tuning process, starting with which features to use as the prediction target.

4.7.2 Dataset Preparation

Despite the noise reduction measures implemented, the final dataset still contained some outliers and unexpected results that could alter the model predictions. A data cleaning phase was added in order to improve the quality of the results.

Table 4.3 summarizes the impact of the various preparation steps on the dataset dimension.

Slower Vanilla

Obfuscation techniques add instructions and complexity to the application, so it is expected that the measured features indicating performance would be bigger than the vanilla ones. If that does not happen, it may be the result of external events, like context switches and CPU migrations, that alter the correct measurements. On the other hand, it is also possible that the execution was unaltered, but the performance degradation was smaller than the measurement precision margin. To guarantee a more consistent dataset, rows where the vanilla metric is bigger than the corresponding obfuscated metric are dropped.

Context Switches and CPU Migrations

Even if the vanilla version was faster than the obfuscated ones, it is still possible that a Context Switch or a CPU Migration affected the accuracy of the measurement, so we also removed rows affected by these events.

Median Deviation Outliers

After analyzing the results, some of the sample measurements still stood out from the other measurements of the same function. While these may represent real data of particular function calls, they are not representative of the common function behavior. For this reason, a filter was implemented to discard all samples that diverged from the median of that function for more than a certain threshold. The median was used for its better outlier-resistance property. The actual threshold depended on the metric analysed and was found experimentally by tuning it to avoid removing too many samples while still providing improved prediction results.

Complete Dataset Samples	Complete Dataset Unique Functions
3,503,892	528

Table 4.2: Complete Dataset size

Cleaning Step	Samples Removed	% Samples Removed
Slower Vanilla	721,855	20.60 %
Context Switches and CPU Migrations	1,768	0.05 %
Median Deviation Outliers	36,210	1.03 %
sweeplist Balancing	1,142,927	32.62 %
luaC_step Balancing	1,037,594	29.61 %

Table 4.3: Cleaning Steps

Filtered Dataset Samples	Filtered Dataset Unique Functions
563,538	472

Table 4.4: Filtered Dataset size

Dataset Balancing

By evaluating the resulting dataset, it was evident that most samples referred to only 2 of the initial 528 functions: `sweeplist` and `luaC_step`. While all of those were actual function calls, it was highly unlikely that those represented over one million different execution paths, so most of them only improved the quality of those function measurements. Keeping only 2000 samples of both functions makes the resulting dataset smaller and more balanced while maintaining high diversity. This benefits from improved training times and better prediction results.

4.7.3 Data splitting

In order to be able to test the ability of the machine learning model to predict the performance of functions, we need to split the data into training and validation data. The model will only train using data from the train dataset, while the data from the validation dataset will be used to evaluate the performance of the complete model. Since it would not make sense to have different calls of the same functions both in the train and validation dataset, the splitting is done at the function level, grouping samples of the same function so that all samples are either in the train, or in the validation dataset.

We tried to balance the two set sizes: ensuring sufficient samples and unique functions in the training set for effective training while also keeping enough samples and unique functions for robust model validation and to avoid overfitting.

The final set sizes are shown in Table 4.5:

Both datasets are also split vertically to divide the data to be used as input and as targets. The train target and validation target will just contain the column that we are

Set	Samples	Unique Functions
Training Set	514,596 (91,32 %)	401 (84,96 %)
Validation Set	48,942 (8,68 %)	71 (15,04 %)

Table 4.5: Training and Validation Sets sizes

trying to predict, which will be the obfuscated counter of a feature that closely relates to performance, or its overhead. In the final models, those will be `cpu-clock_opaque` (overhead) and `cpu-clock_flatten` (overhead).

The train input and validation input will contain all columns available to the developers who want to use the model: vanilla static and dynamic metrics. In a real scenario, these could easily be computed automatically in a Continuous Integration / Continuous Delivery pipeline without effort from the programmer and with little time required. With only those metrics and access to the pre-trained machine learning model, predicting the performance degradation of all obfuscation techniques covered by the framework becomes instantaneous.

Note that the `region` and `seed` columns must be removed since they are the only non-numerical features, so they cannot be used for model training.

4.7.4 Model tuning

Prediction target

Since many metrics are indicators of the applications' performance, it is not straightforward to decide which of them will be easier and more valuable to predict. The selected candidate obfuscated features to be predicted are: `avg_time-elapsed_ms`,`cpu-clock`,`cpu-cycles`,`ref-cycles`,`instructions`. For each of those, also the corresponding Overhead was evaluated. It is measured as:

$$Overhead = \frac{Obfuscated\ Metric}{Vanilla\ Metric}$$

After conducting multiple tests, the metric that resulted in better-performing models was the `cpu-clock` overhead.

Dataset preparation configuration

Since it was initially unclear which filters would improve the dataset for better results, many tests were performed with the various filters enabled or disabled. The percentage deviation from the median was also adjusted according to the obfuscation technique used, resulting in a 250% deviation for both Flatten and Opaque. The split size was also tuned so that the validation set would contain between 7% and 30% of all samples and about 15% of all distinct functions.

Features engineering

One of the aspects that need to be taken into consideration when dealing with machine learning models is the curse of dimensionality phenomenon. According to the statement, if the amount of available data does not grow exponentially as the number of features increases, performance may decrease. This happens because predicting the target variable requires finding patterns and relationships between features and that variable. With high-dimensional data, grouping data points together becomes difficult, as they appear sparse and dissimilar.

Feature engineering consists of identifying which features are the most useful, relevant and impactful on model performance and excluding those that are not. This helps to focus on the features that matter without getting distracted by the irrelevant ones. The added benefit is that with a smaller dataset, the training time will also be reduced. To decide which features are less relevant, two approaches can be used [26]:

- `model.feature_importances_`: This is a built-in attribute provided by some tree-based models like the Random Forest Regressor. It calculates the importance based on how much a feature splits a node in the decision tree, leading to a reduction in impurity. The problem with this approach is that it favours features with high cardinality, which are common in our dataset.
- `permutation_importance`: This is a more general technique that can be applied to any model, not just tree-based ones. It works by shuffling the values of a single feature and then measuring the decrease in model performance. The more the performance drops, the more important the feature is considered to be. It takes longer to compute, but it avoids the bias towards high-cardinality features. This is the technique that was chosen.

To identify the most important features, one model for each obfuscation technique was trained using all available features and with the `permutation_importance` technique. The results are shown in 4.7 and 4.9. Various feature removal strategies were explored to determine which combinations would lead to higher performance improvements. Ultimately, features with importance values below 0.01 were selected for removal from the dataset, resulting in the final models seen in 4.8 and 4.10. The opaque chart is on a logarithmic scale to improve the visualization of the impact of various features. This scale prevents the `cycle_cost` and `jump` features from dominating the visual representation and allows for a better comparison of all features' influence on the model's accuracy. To improve visual clarity, dynamic features are indicated in blue, while static features are indicated in red.

4.7.5 Random Forest Regressor

The library that we decided to use is Scikit-learn, one of the most popular machine-learning libraries in Python. The `RandomForestRegressor` object implements the model that we are going to use. [27]

Hyperparameter tuning is not as effective as for other models, so we only tuned these:

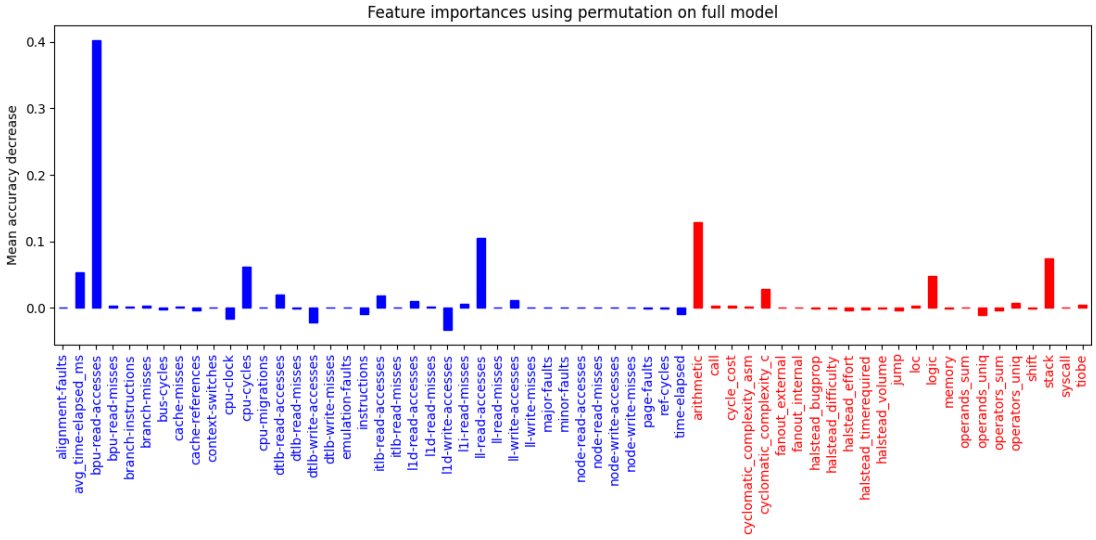


Figure 4.7: Features importance for Control Flow Flattening

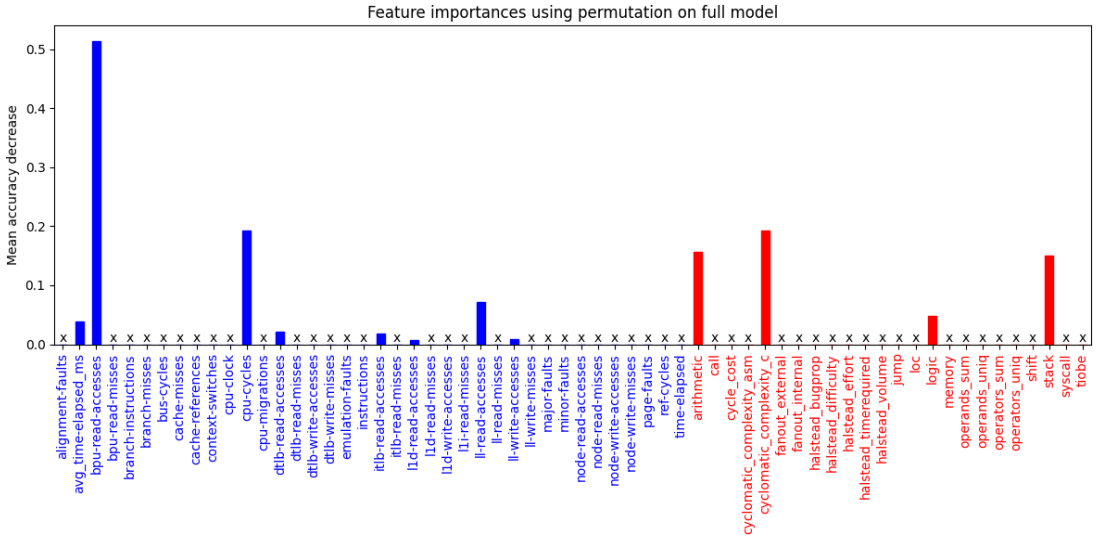


Figure 4.8: Features importance for Control Flow Flattening after Feature Selection

- `n_estimators`: The number of trees in the forest. A higher number typically produces better results at the expense of higher training time. Different values were used when exploring different configurations (25) and when trying to improve the models (100).
- `n_jobs`: How many threads to run in parallel to speed up training. Typically used 4 to avoid saturating the machine.
- `random_state`: Control randomness in sample selection to create new training sets for each tree in the forest (bootstrapping). Also, randomness can be controlled by

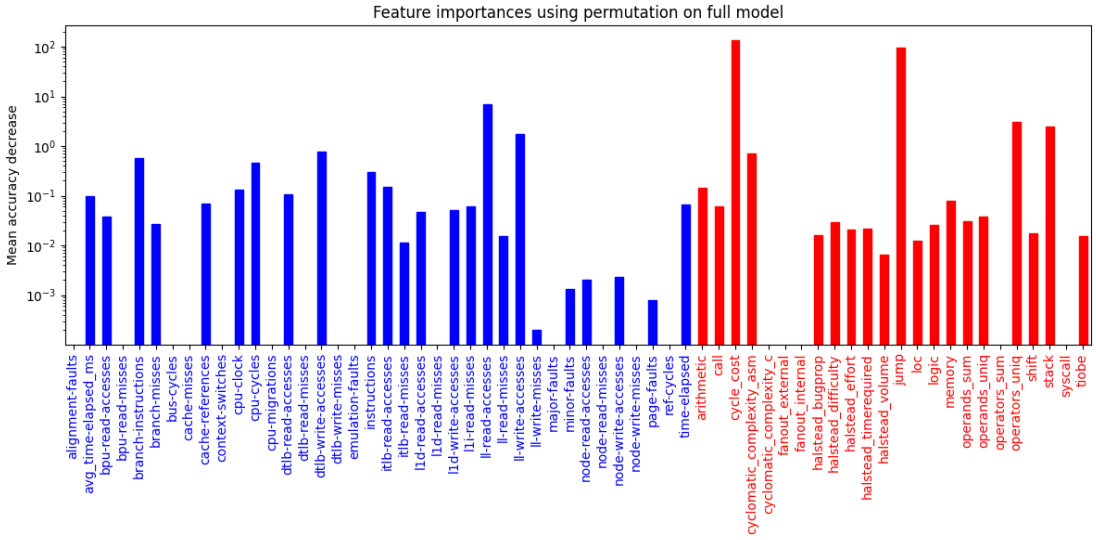


Figure 4.9: Features importance for Opaque Predicates

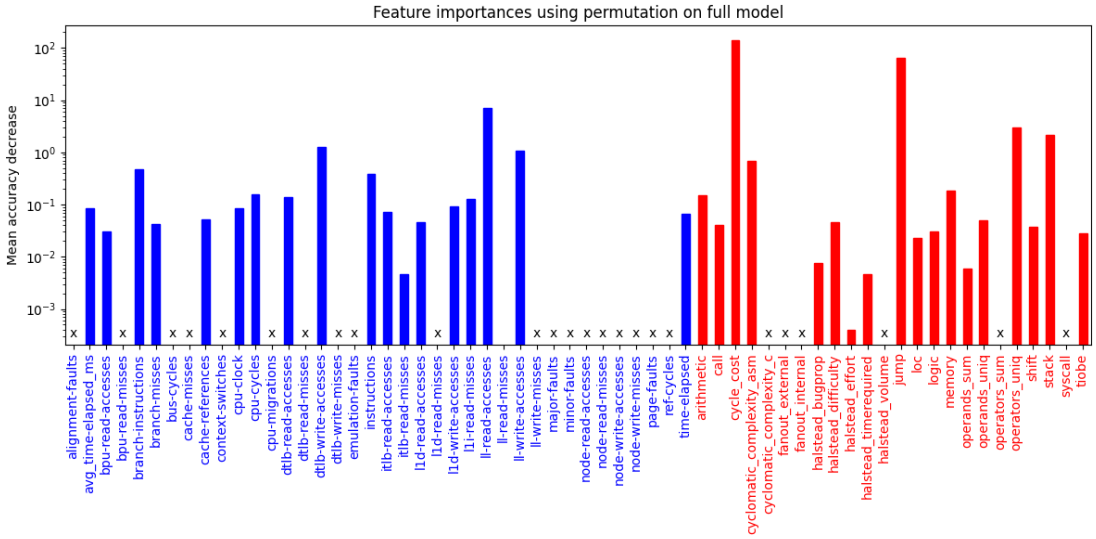


Figure 4.10: Features importance for Opaque Predicates after Feature Seletion

selecting a subset of features to create the best split point. Guarantees reproducibility of training.

4.7.6 Neural Network Regressor

Because of their powerful and flexible nature, neural networks were used to try to get different models with better results. Because of its simplicity of use, the Scikit-learn's

neural_network library was used. This provides a Multi-Layer Perceptron in the form of the MLPRegressor model, which is the type of neural network that we will use. [28] The disadvantage of this library is that it is slower than other more complex alternatives because it does not take advantage of GPU-based implementations.

To reduce training time, an initial analysis was performed using only 10% of the full dataset in order to understand if the results were promising enough to scale to a bigger dataset. Since the results were optimistic, the training was repeated on the complete dataset.

Hyperparameters

One of the disadvantages of neural networks is that they require extensive targeted hyperparameters tuning to achieve good results. Hyperparameters are configuration settings that regulate how the neural network is internally structured and its learning process.

Since no literature exists for using neural networks to predict performance, we could not use generally good values for the hyperparameters, so we tested several [29]:

- `hidden_layers`: These determine the complexity of the model. More layers can potentially capture more complex relationships in the data, but also increase training time and risk overfitting. Tested values used were 1,2,4,8,16
- `neurons`: These define the width of the network. More neurons allow for more complex decision boundaries, but can also lead to overfitting. Tested values used were: 16,32,64,128,256,512,1024
- `learning_rate_init`: The initial learning rate to use. It controls the step size when updating the weights. This could have a big impact on the model's performance. Tested values used were: 0.001, 0.01, 0.05, 0.10
- `random_state`: Guarantees reproducibility in the neural network training by reusing the same seed when random operations are needed. It ensures consistent weight initialization to neuron connections and consistent batch sampling of the training data during a single training iteration. The value used was 42

Scalers

Multi-layer Perceptrons are sensitive to the scale of the data, as having vastly different ranges can negatively impact the training process. Many machine learning algorithms assume feature values centered around a common value and with similar scales. Without these assumptions, the performance of the model may degrade, with the risk of not converging to an optimal solution.

Scaling the data is highly recommended to avoid this. This means adding or subtracting a constant and then multiplying or dividing by a constant so that the resulting values

are centered around 0. Scikit provides several Scalers that transform the data in different ways. The two tested are [30]:

- StandardScaler: This is a common scaling technique that centers the data by subtracting the mean of each feature and then scales it by dividing by the standard deviation. This results in features with a mean of 0 and a standard deviation of 1. [31]
- RobustScaler: This scaler is designed to be more robust to outliers. Instead of using the mean and standard deviation, it subtracts the median of each feature and then scales it by dividing it by the interquartile range (IQR). The IQR represents the range between the 25th and 75th percentile of the data. Both median and IQR are more resistant to outliers than average and standard deviation. [32]

Different scalers could be applied to the training and validation inputs and targets, but since they all contain similar data, the same scaler was used. This means that the resulting predictions will also be scaled, so an additional operation to invert the scaler's transformation is required before evaluating the model. [33]

4.7.7 Model Evaluation

To measure regression performance, metrics described in Section 3.2.6 were used. The Scikit-learn library already implements functions to compute most of the metrics needed, like RMSE, MAE, MAPE, R-squared and Percentiles.

We decided to include additional metrics to have a practical idea of how far the predictions were from the actual values. To achieve this, we counted the number of samples with Prediction Errors $< 5\%$, $< 10\%$, $< 20\%$, $< 30\%$, $> 30\%$ and reported how many samples belonged to each group.

Chapter 5

Results

This chapter summarizes the results of this thesis work.

5.1 Configuration used

The dedicated machine Hardware and Software versions used by the dedicated machine are listed here:

```
CpuModel = Intel Core i7-4910MQ
OSVersion = "Ubuntu 21.04"
KernelVersion = "5.8.0-59-generic"
GLIBVersion = "Ubuntu GLIBC 2.33-0ubuntu5"
CompilerVersion = "gcc (Ubuntu 10.3.0-1ubuntu1)"
python = "^3.10"
r2pipe = "^1.8.0"
multimetric = "^2.0.5"
Pygments = "^2.16.1"
chardet = "^5.2.0"
tqdm = "^4.66.1"
pandas = "^2.1.0"
scikit-learn = "^1.3.0"
```

5.2 Overhead Distribution

Table 5.1 summarizes the overhead distribution for the two obfuscation techniques used. For most samples, the impact of obfuscation on performance is small, but there are cases where it is significant. In these cases, the control flow flattening technique significantly impacts the performance.

Percentile	Flatten	Opaque
25th percentile	0.85%	0.82%
50th percentile	1.97%	2.23%
75th percentile	5.63%	5.77%
90th percentile	21.57%	9.59%

Table 5.1: Overhead Distribution

5.3 Random Forest Regressors Evaluation

Different simple regressors were created to serve as benchmarks for the performance of our models. These use simple prediction strategies derived from the filtered dataset:

- Average: This regressor uses the average of the `cpu-clock` overhead of the filtered dataset to make all predictions.
- Average of Average: This regressor calculates the average overhead for each function of the filtered dataset, and then uses the average of those averages for all predictions.
- Weighted Average: This regressor calculates a weighted average of the overhead based on the frequency of each function in the filtered dataset.

Two of the obtained models are compared with the benchmarks:

- All Features: This is the Random Forest Regressor model obtained from using all the features available in the dataset
- After Feature Selection: This is the final Random Forest Regressor model trained solely on the remaining features after removing the less important ones

Tables 5.2 and 5.3 present the comparison between the evaluation metrics values of the main models.

For both obfuscation techniques, the data show that feature selection improved all the metrics compared to the model trained on all features. Removing irrelevant or redundant features results in a model that generalizes better and makes more accurate predictions.

From now on, we will only consider the results from the models after feature selection, since they represent the most promising models.

All three error metrics (Root Mean Square Error, Mean Absolute Error, and Mean Absolute Percentage Error) show significant improvement compared with the benchmark regressors. Their values decrease considerably, indicating a reduction in prediction errors. The RMSE is inferior to the standard deviation of the validation set, indicating that the model is uncovering significant patterns in the data, resulting in more accurate predictions than simply using the average values. The error prediction reduction is more evident in the Control Flow Flattening model. This is likely because this obfuscation technique introduces higher variability in possible overhead values, as the higher standard deviation

Metric	Average	Average of Average	Weighted Average	All Features	After Feature Selection
Standard Deviation of Validation Set	0.1821	0.1821	0.1821	0.1821	0.1821
Root Mean Square Error	0.2047	0.2200	0.2255	0.1212	0.1049
Mean Absolute Error	0.1671	0.1906	0.1983	0.0584	0.0469
Mean Absolute Percentage Error	14.91 %	17.30 %	18.07 %	4.63 %	3.96 %
R-squared	-0.2632	-0.4602	-0.5342	0.5572	0.6682
Samples with Prediction Error <5%	4.66 %	4.06 %	3.94 %	78.16 %	81.58 %
Samples with Prediction Error <10%	13.72 %	9.88 %	8.75 %	88.90 %	90.72 %
Samples with Prediction Error <20%	94.92 %	80.82 %	59.75 %	95.89 %	96.40 %
Samples with Prediction Error <30%	97.91 %	98.12 %	98.16 %	97.97 %	98.98 %
Samples with Prediction Error >30%	2.09 %	1.88 %	1.84 %	2.03 %	1.02 %

Table 5.2: Models evaluation for Control Flow Flattening Obfuscation

indicates. Consequently, the average predictions for the Opaque model tend to be closer to the actual values compared to the Control Flow Flattening model.

On the other hand, the R-squared metric in both cases goes from negative to positive and shows a substantial increase. This indicates a strong improvement in the ability of both models to explain the variance in the target variable, regardless of the obfuscation technique used. Interestingly, despite a higher standard deviation, the Control Flow Flattening model achieves an R-squared value even higher than the Opaque model. This can be explained by the nature of Random Forest models, which can handle complex and varied datasets. In this case, the higher variability might help the model distinguish between data points with similar values. It can also provide a broader range of patterns and relationships in the data for the Random Forest to learn from, resulting in a more robust model that generalizes better to unseen data.

Two main chart types can help us visualize the actual model behavior effectively: histograms and scatterplots.

Histograms allow us to see the distribution of actual and predicted sample values across different overhead value ranges. A logarithmic scale was used to improve the visualization of bins of different sizes.

When comparing our model to the average regressor, it is evident that the number of samples predicted in each overhead range closely matches the distribution of the actual

Metric	Average	Average of Average	Weighted Average	All Features	After Feature Selection
Standard Deviation of Validation Set	0.0953	0.0953	0.0953	0.0953	0.0953
Root Mean Square Error	0.0953	0.1167	0.0992	0.0831	0.0724
Mean Absolute Error	0.0506	0.1002	0.0446	0.0425	0.0408
Mean Absolute Percentage Error	4.38 %	9.35 %	3.70 %	3.81 %	3.66 %
R-squared	-0.0000	-0.4989	-0.0824	0.2397	0.4221
Samples with Prediction Error <5%	81.65 %	10.64 %	88.00 %	81.19 %	81.34 %
Samples with Prediction Error <10%	93.37 %	49.00 %	92.34 %	92.38 %	92.30 %
Samples with Prediction Error <20%	96.37 %	97.62 %	95.86 %	97.89 %	98.22 %
Samples with Prediction Error <30%	98.80 %	99.57 %	98.35 %	99.32 %	99.63 %
Samples with Prediction Error >30%	1.20 %	0.43 %	1.65 %	0.68 %	0.37 %

Table 5.3: Models evaluation for Opaque Obfuscation

overhead values, both in the Flatten (Figures 5.1 and 5.2) and in the Opaque (Figures 5.3 and 5.4) models.

While histograms give us a general overview of the ranges of prediction distributions, scatterplots offer a more detailed analysis by revealing the relationship between individual sample overhead values and their corresponding predictions. However, a static scatterplot can become cluttered when dealing with tens of thousands of samples. The ‘plotly‘ Python library allows interactive charts that allow zooming and panning, granting the ability to explore different regions of the graph.

A static image of this chart can still be helpful to gain information. These scatterplot components are:

- Axes: The x-axis represents the actual overhead values, and the y-axis represents the predicted overhead values.
- Perfect Predictions: Points lying on the diagonal blue line indicate perfect matches between the predicted and actual values.
- Prediction Error: The two red lines, positioned 10% above and below the diagonal line, represent a prediction error of +/- 10%. Points falling between these lines have predictions within 10% of the actual value.

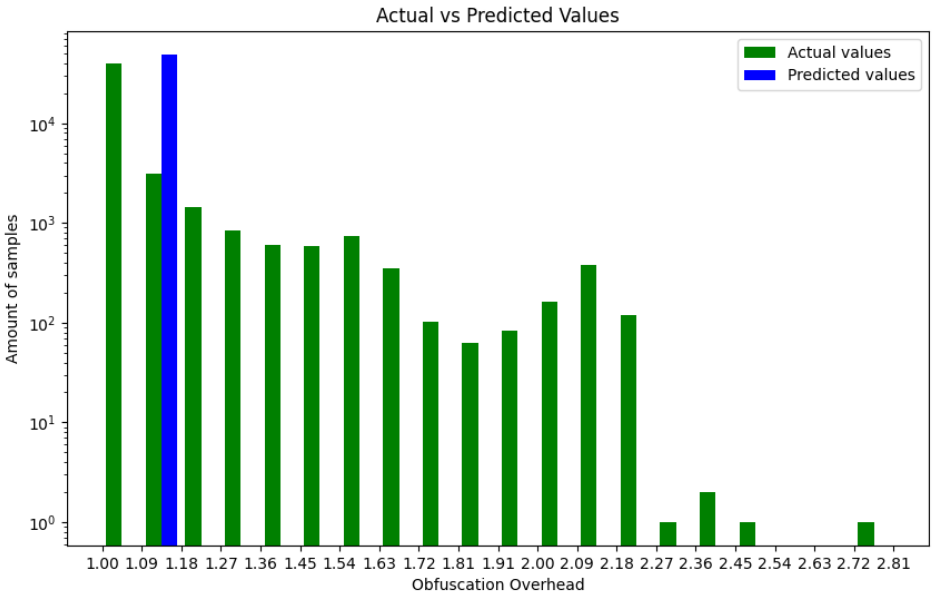


Figure 5.1: Flatten average regressor histogram

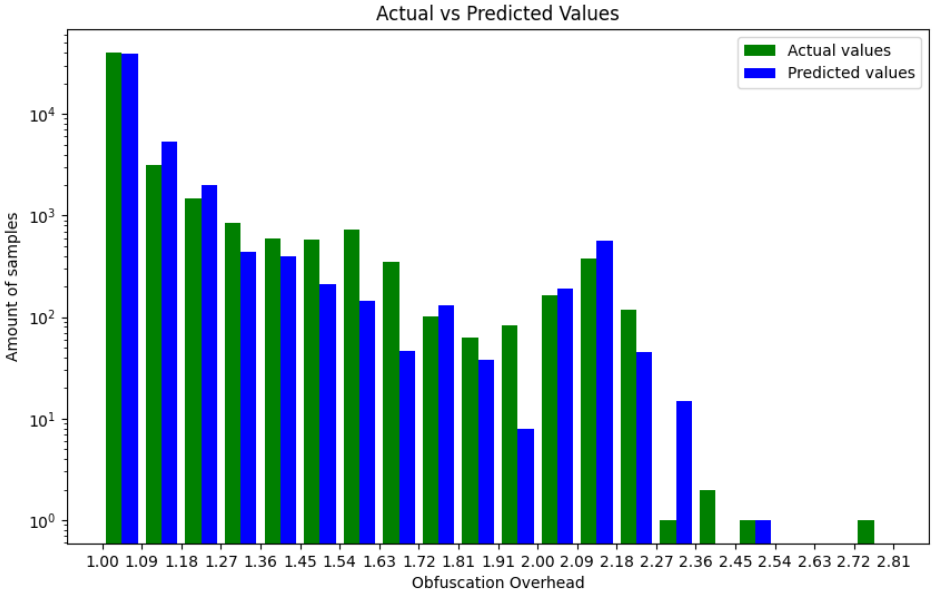


Figure 5.2: Flatten model histogram

- Color-coding: The color intensity of a data point reflects its distance from the diagonal

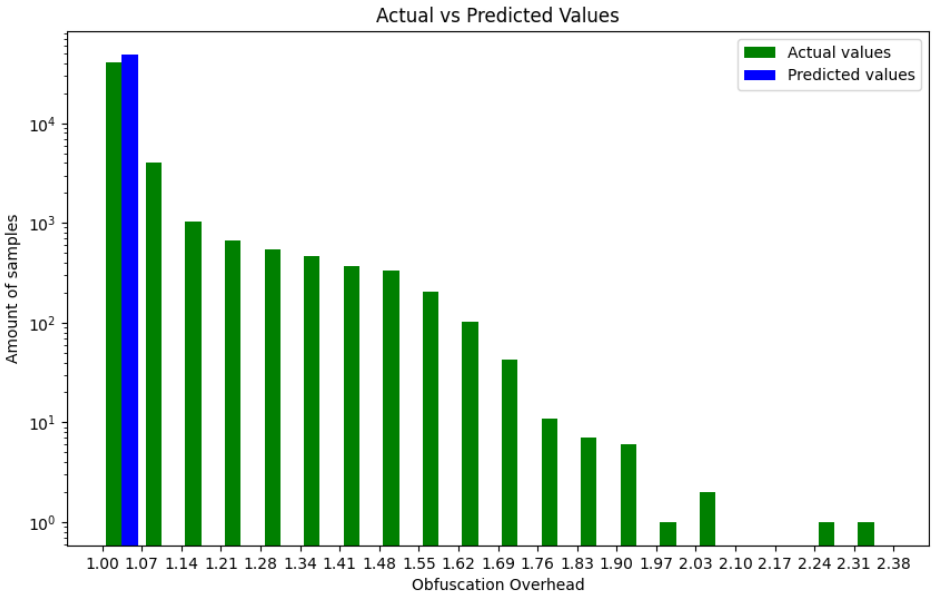


Figure 5.3: Opaque average regressor histogram

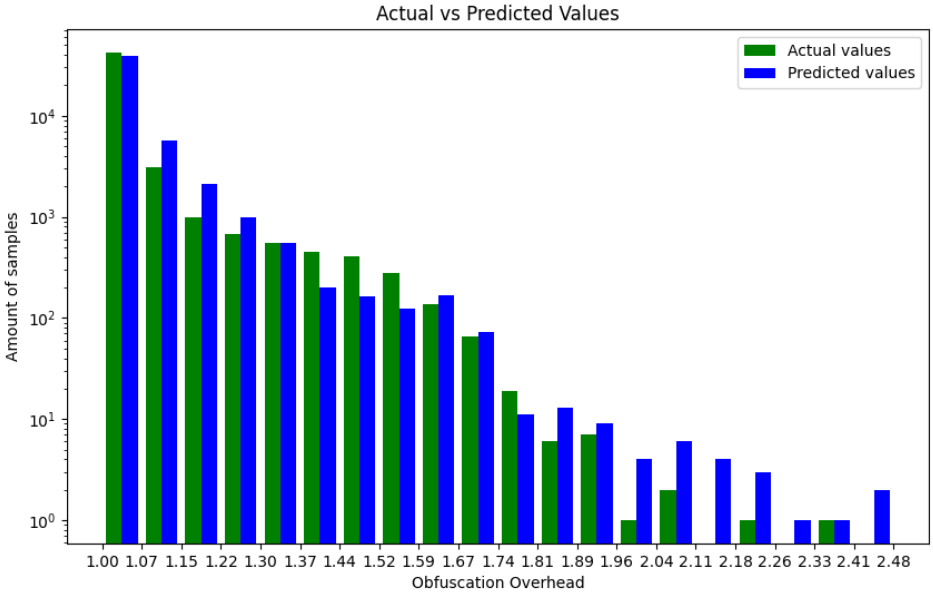


Figure 5.4: Opaque model histogram

line. Points with colors closer to green are closer to perfect predictions, while points

with colors closer to red have larger prediction errors.

- Histogram Integration: Histograms alongside the scatterplot can reveal the overall distribution of the data points, highlighting which areas have higher concentrations of samples. Note that the prediction histograms for the average regressors are represented unintuitively by the application since they actually correspond to a single prediction value while it looks like a large distribution.

Figures 5.5 and 5.7 show visually that even if the average benchmarks may obtain a seemingly good evaluation for some metrics like Prediction Errors, they actually don't follow the sampling trend at all.

Figure 5.8, and even more Figure 5.6 show that predictions are not casual, but tend to follow the absolute overhead values even when they are very distant from the average of the distribution.

Ultimately, our meticulous efforts to refine the framework workflow, incorporate new applications, extract static features, implement noise reduction techniques, establish a precise dynamic feature extraction system, clean the dataset, tune and evaluate the models ultimately paid off. This dedication resulted in high-quality predictions, demonstrating a significant improvement from the research's initial stages.

5.4 Neural Network Regressor Evaluation

The same feature selection used for the Random Forest models was applied to the Neural Network models to compare the two algorithms' performance more consistently. After tuning the network's hidden layers, number of neurons and initial learning rate, Neural Network models results were compared to find the best.

Graphs in Figure 5.9 only focus on the RMSE value obtained by the models using different hyperparameters values. Other evaluation metrics are not shown since having a high RMSE compared to the validation set's standard deviation directly correlates to poor model performances. The figures also contain a red line corresponding to the Standard Deviation of the Validation Set, which is used as a baseline value to compare the RMSE. RMSEs lower than this threshold indicate promising models, for which other evaluation metrics need to be analyzed.

While none of the Flatten models achieved performance comparable to the Random Forest Models (Table 5.4), one promising model was obtained with the Opaque model, using an Initial Learning Rate of 0.01, 8 layers and 64 neurons.

This promising result led to exploring other Initial Learning Rate values for the Opaque model, as seen in Figure 5.10.

This revealed another promising model, with an Initial Learning Rate of 0.001, 4 layers and 32 neurons.

A comparison of their performance with the Average benchmark model and the final Random Forest model can be seen in Table 5.5.

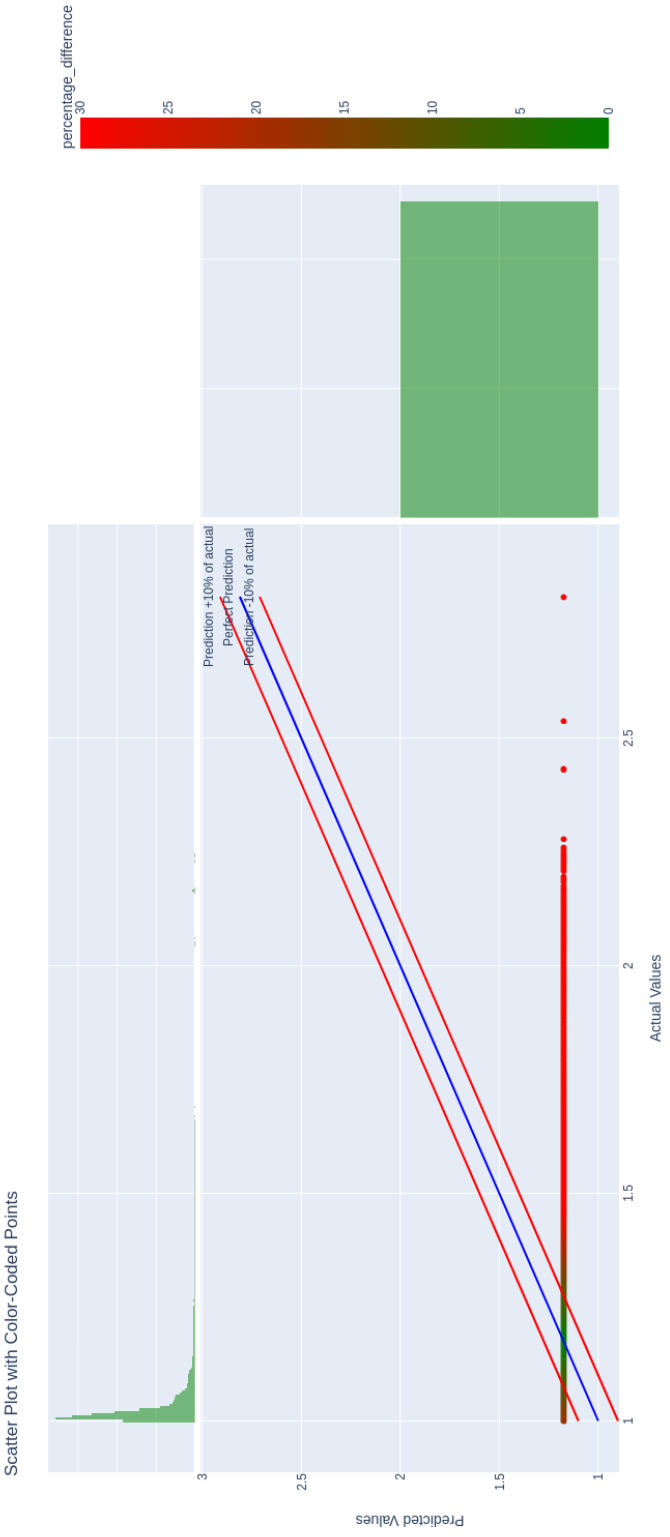


Figure 5.5: Flatten average regressor scatterplot

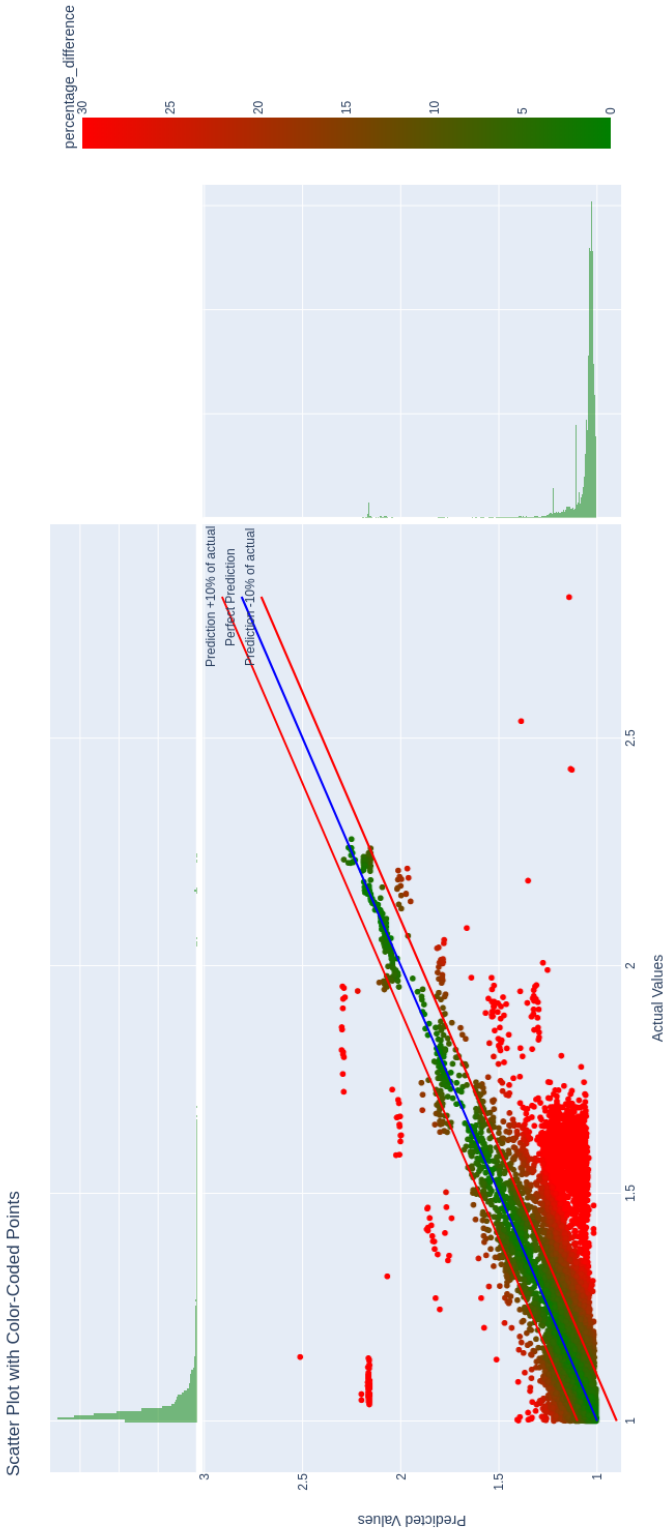


Figure 5.6: Flatten model scatterplot

Metric	**Average**	**Random Forest**	**Neural Network (8,64) 0.01**
Standard Deviation of Validation Set	0.1821	0.1821	0.1821
Root Mean Square Error	0.2047	0.1049	0.2097
Mean Absolute Error	0.1671	0.0469	0.1753
Mean Absolute Percentage Error	14.91 %	3.96 %	15.74 %
R-squared	-0.2632	0.6682	-0.3262
Samples with Prediction Error <5%	4.66 %	81.58 %	4.47 %
Samples with Prediction Error <10%	13.72 %	90.72 %	12.17 %
Samples with Prediction Error <20%	94.92 %	96.40 %	95.07 %
Samples with Prediction Error <30%	97.91 %	98.98 %	98.01 %
Samples with Prediction Error >30%	2.09 %	1.02 %	1.99 %

Table 5.4: Neural Network model evaluation for Control Flow Flattening Obfuscation

The final model outperforms all other Neural Network models, with results comparable to the best Opaque Random Forest Regressor model. While the Random Forest Regressor ultimately emerged as the superior model, having lower RMSE, higher R-squared and lower overall prediction errors, the final Neural Network model demonstrates a marginally improved MAE and MAPE.

Metric	**Average**	**Random Forest**	**Neural Network (8,64) 0.01**	**Neural Network (4,32) 0.001**
Standard Deviation of Validation Set	0.0953	0.0953	0.0953	0.0953
Root Mean Square Error	0.0953	0.0724	0.0841	0.0817
Mean Absolute Error	0.0506	0.0408	0.0527	0.0396
Mean Absolute Percentage Error	4.38 %	3.66 %	4.80 %	3.53 %
R-squared	-0.0000	0.4221	0.2220	0.2648
Samples with Prediction Error <5%	81.65 %	81.34 %	76.10 %	79.83 %
Samples with Prediction Error <10%	93.37 %	92.30 %	86.78 %	91.35 %
Samples with Prediction Error <20%	96.37 %	98.22 %	96.03 %	97.53 %
Samples with Prediction Error <30%	98.80 %	99.63 %	99.89 %	99.64 %
Samples with Prediction Error >30%	1.20 %	0.37 %	0.11 %	0.36 %

Table 5.5: Neural Network models evaluation for Opaque Obfuscation

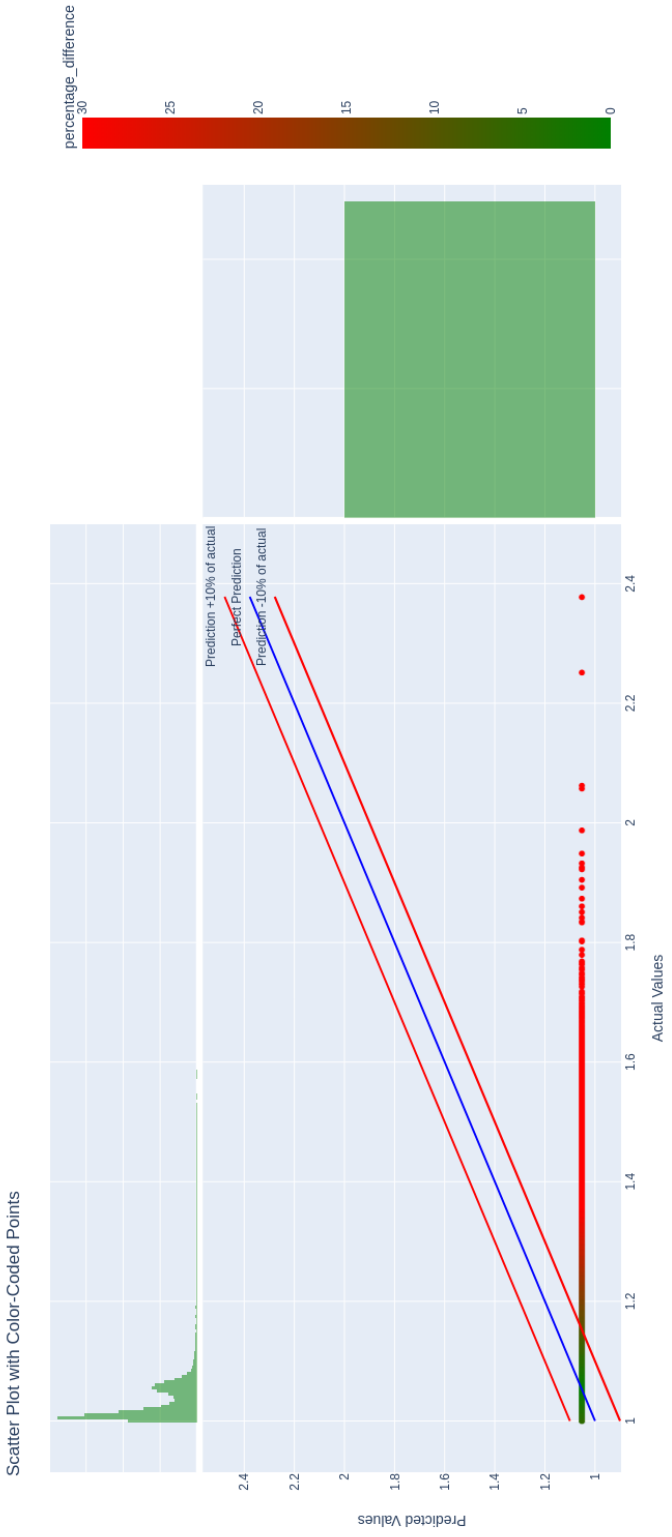


Figure 5.7: Opaque average regressor scatterplot

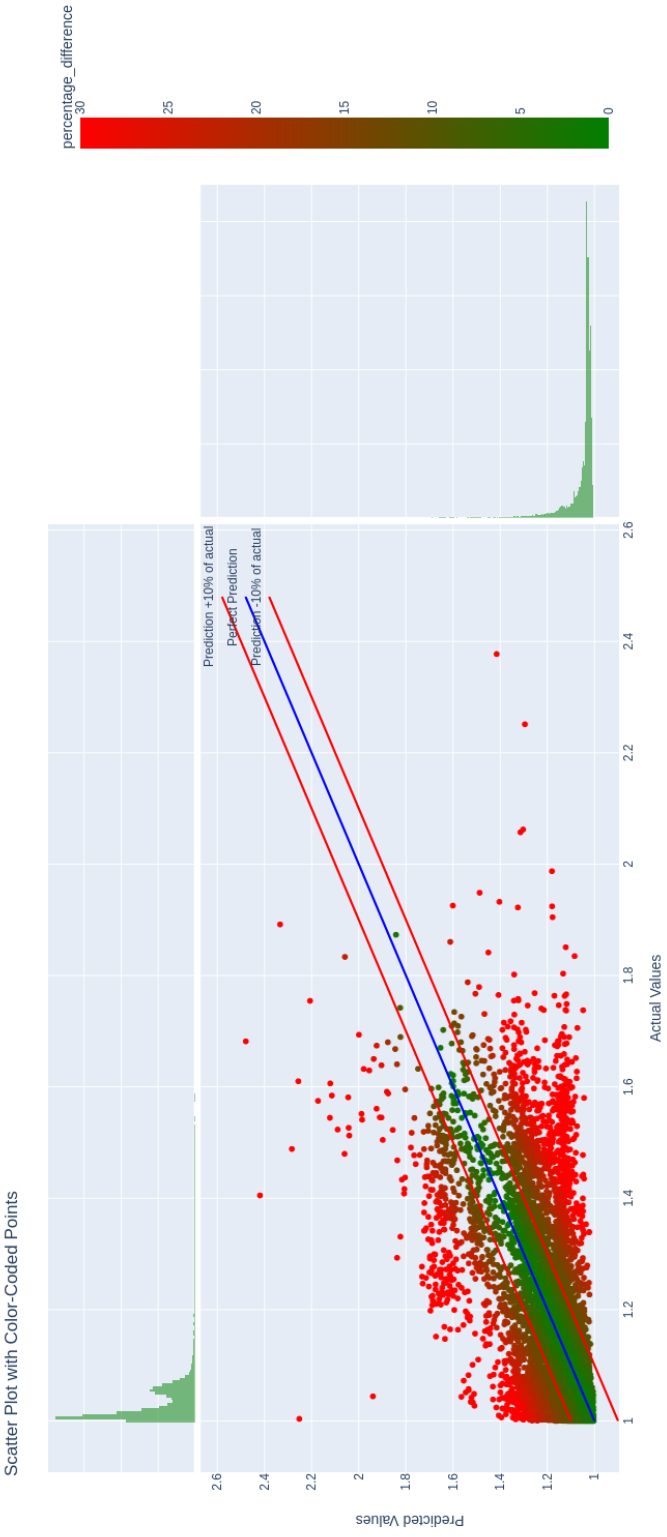
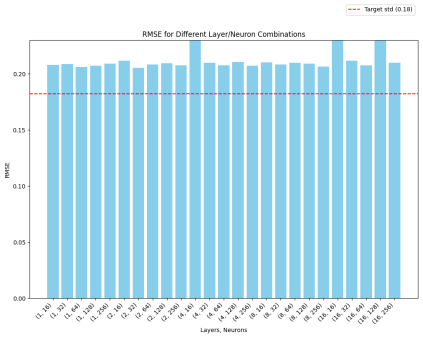
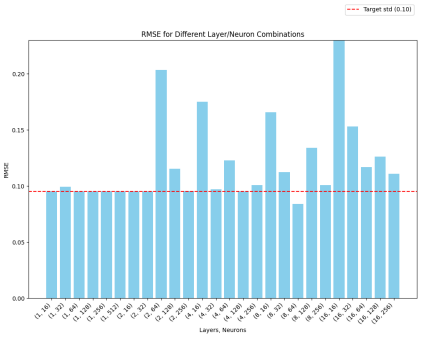


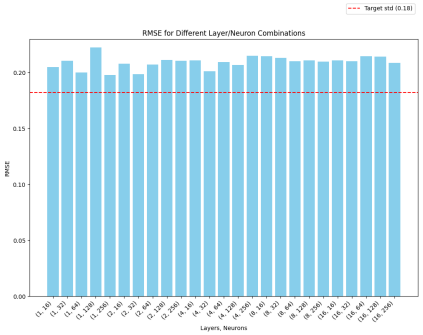
Figure 5.8: Opaque model scatterplot



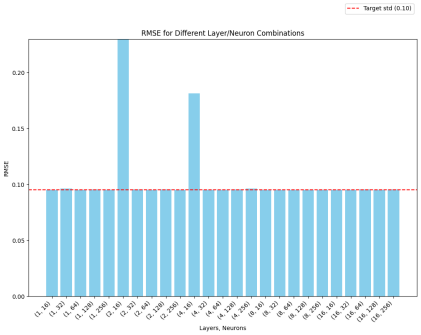
(a) Flatten 0.01 Initial Learning Rate RMSE



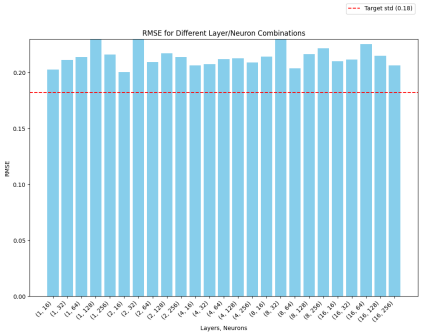
(b) Opaque 0.01 Initial Learning Rate RMSE



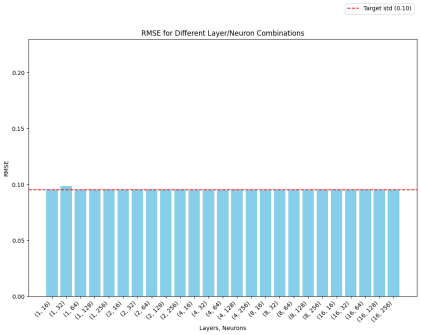
(c) Flatten 0.05 Initial Learning Rate RMSE



(d) Opaque 0.05 Initial Learning Rate RMSE

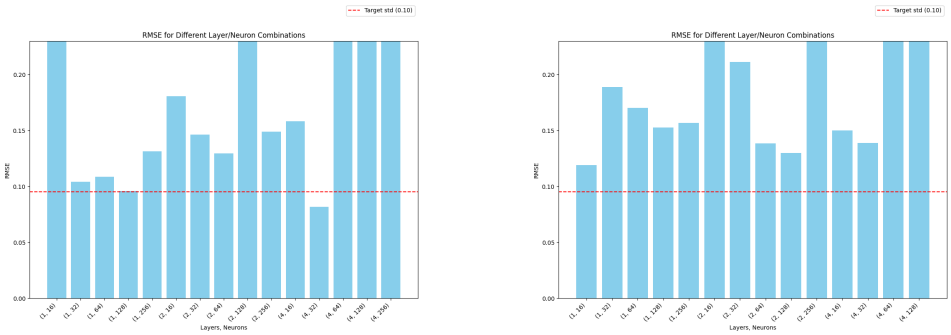


(e) Flatten 0.1 Initial Learning Rate RMSE



(f) Opaque 0.1 Initial Learning Rate RMSE

Figure 5.9: Models RMSE values with different hyperparameters



(a) Opaque 0.001 Initial Learning Rate RMSE

(b) Opaque 0.0001 Initial Learning Rate RMSE

Figure 5.10: Opaque models RMSE values with additional hyperparameters

Chapter 6

Future Works

The meticulous enhancements made to the various aspects of the framework workflow have resulted in significant improvements towards a more accurate prediction of the performance overhead introduced by obfuscation techniques. Future research can expand upon these robust findings to explore additional areas and integrate new aspects. Some of the possible improvement areas are listed in this section.

6.1 Application selection

Since the goal is to generalize the prediction of the performance overhead to all possible functions, it would be beneficial for the models to increase data diversity. This can be achieved by adding more applications to the pool or by considering more execution paths of the current applications.

Adding new applications is a challenging process; it requires proper configuration of the application and its build settings to ensure compatibility with Tigress obfuscation.

Increasing the coverage of the applications already present in the dataset can be done by fuzzing or manually creating new tests. Both options require enough knowledge of the application to create useful tests that would increase the execution path coverage, without adding redundant data to the dataset. A possible solution would be to let all interested developers themselves add their application and its testing inputs to the pool.

6.2 Obfuscation

Since only Control Flow Flattening and Opaque Predicates were explored, a necessary next step will be to apply different obfuscation techniques and their combinations to represent real obfuscation scenarios better.

6.3 Machine learning

Due to the sensitivity of Neural Network models to hyperparameters, it is possible that improved models could be discovered through a more extensive and comprehensive tuning process.

A different approach can be explored, based on training multiple models and then using the most appropriate one depending on the function that needs to be obfuscated. The other models would be trained on different sets of functions, depending on their properties. Several criteria can be used to partition functions based on the available static and dynamic features. Top Down Microarchitecture Analysis is one possible approach commonly used to find workload bottlenecks in the CPU microarchitecture, but it could be leveraged to cluster functions according to their bottleneck location. For example, a model could be trained on functions limited by many branch mispredicts, while another could be trained on functions limited by many cache misses, and so on. This could help train more specialized models with better performances and be used on different types of functions.

6.4 Protection Strength

While this framework addresses the problem of predicting the performance impact of obfuscation, the other important factor to consider when choosing which technique to use is the protection strength that each technique provides. This is a very complex aspect to consider and estimate since modeling MATE attacks is very difficult. Some promising recent advancements in the literature explore ways to evaluate this through formal approaches, user studies, code metrics or resistance against symbolic execution attacks [34]. Integrating together the performance degradation prediction and the protection strength for each obfuscation technique would empower the application developers to make informed decisions and effortlessly find the best tradeoff between security and performance.

Appendix A

User Manual

A.1 Directory Structure

- app/: Applications files
  - source/: Applications sources
    - `<application-name>`/
  - seeds/: Applications inputs
    - `<application-name>`/
- config/: Configuration files
  - compiling_dict.json
  - config.ini
- workdir/: Default folder for ‘performance.py‘ outputs
  - app/: Obfuscated sources and executables
  - results/: Feature extraction results
    - `<application-name>`/: Static metrics extracted and Dynamic features extracted using Event Counting
      - sampling/: Dynamic features extracted using Event Sampling
- utils/: Compiled third party utilities
- output_analysis/: Output manipulation scripts
  - csv_processing/: Scripts to manipulate csv files, like merging
  - ML/: Machine Learning related scripts
- benchmark_system_config.py

- measure.py
- metrics.py
- performances.py
- README.md
- requirements.txt
- utils.py

A.2 Environment setup

In order to install all needed dependencies, a `README.md` file is included, containing the required steps:

```
# clone repository
git clone https://github.com/b3xul/2023_Obfuscation_Thesis.git
cd 2023_Obfuscation_Thesis

# install system dependencies
sudo apt-get install build-essential python3 virtualenv pmccabe cpuset
    libreadline-dev

# create virtualenv using Python version >= 3.10
virtualenv -p $(which python3.10) PyEnv
source PyEnv/bin/activate

# install python libraries (r2pipe multimetric Pygments chardet tqdm
    pandas scikit-learn matplotlib plotly)
pip install -r requirements.txt

cd utils

# add environmental variables for tigress
export TIGRESS_HOME=$(pwd)/tigress/3.1
export PATH=$PATH:$(pwd)/tigress/3.1

# install radare2
git clone https://github.com/radareorg/radare2
radare2/sys/install.sh

# Test that the external dependencies work:
ctags-bin/ctags --version

perforator-0.4.1/perforator -v

r2 -v

```

```
tigress --Environment=x86_64:Linux:Gcc:4.6 --Transform=Virtualize --
    Functions=main,fib,fac --out=result.c $(pwd)/tigress/3.1/test1.c
gcc -o result.exe result.c
strip result
./result.exe
rm a.out result.c result.exe

cd ..
```

A.3 Add new application

Adding a new application to the dataset requires adding its source code, including its `Makefile` under `app/source/<application-name>`. All input files required for testing needs to be added under `app/seeds/<application-name>`. An entry describing the application must be added in the `config/compiling_dict.json` file.

```
"<application-name>": {
        "comment": "Insert here a brief description of the app",
        "tigress_flags": "Compilation flags that may be needed for
    Tigress compatibility",
        "compile_cmd": "Gcc parameters needed for compiling this
    specific app. $IN_FILE and $OUT_FILE are used as placeholders for
    the input .c files and the output .exe file",
        "args": "Execution arguments for the app. $SEED is a placeholder
     that will refer to files under app/seeds/<application-name>",
        "cleanup_cmd": "Bash command to be executed after each run to
    remove leftover files. CAUTION!!! ANY COMMAND inserted here will be
    executed after each run. Use at your own risk.",
        "git": "If available, reference to the source URL of the app"
    },
```

An example entry for the `fastdnaml` application will look like this:

```
    "fastdnaml": {
        "comment": "ML applied to DNA analysis. Heavly modified to
    delete conditional compiling for Master/Slave config. Preserved
    Sequential processing ",
        "tigress_flags": "",
        "compile_cmd": " -O2 -fno-inline-small-functions -fno-indirect-
    inlining -fno-inline-functions  -fno-inline-functions-called-once -
    fno-early-inlining  $IN_FILE -o $OUT_FILE -lm",
        "args": " < $SEED",
        "cleanup_cmd": "rm -f checkpoint* treefile*",
        "git": "https://directory.fsf.org/wiki/FastDNAml"
    },
```

A.4 Configure features extraction

The `config/config.ini` file is used to store the configuration information needed by the `performances.py` script to perform features extraction. It is divided in multiple sections: The `Paths` section contains paths used by the various external dependencies, inputs and outputs

```
[Paths]

Tigress_path = ./utils/tigress/3.1/
Ctags_path = ./utils/ctags-bin/ctags
Perforator_path = ./utils/perforator-0.4.1/

Sources_path = ./app/source/
Seeds_path = ./app/seeds/
Work_path = ./workdir/
```

The `Paths` section contains different options that can be enabled and disabled by setting them as `true` or `false` to customize the `performances.py` behavior

```
[Processing]
Clean = # Clean workdir before processing
Merge = # Merge .c files using tigress -merge
Build = # Compile the apps
Obfuscate = # Enable obfuscation
CollectMetrics = # Collect Static Metrics
MeasureTime = # Event Counting
MeasureSample = # Events Sampling

PostExecCleanup = # Clean residual files after execution
SkipIfError = # Skip the app if an exception occurs
DeleteOut = # Delete .out files in root after compilation
```

The `Compile` section includes compilation flags that will be applied to all applications measured, in addition to the ones present in the `compiling_dict.json` entry of each application. It also allows to set the verbosity of the tigress commands.

```
[Compile]
default_compiler = # Default compiler command
tigress_merge_cmd = # Tigress_merge parameters
default_compile = # Default compiler arguments
```

```
debug_flags = # Debug compiler flags
fixed_flags = # Mandatory compilation flags

tigress_verbosity = false
```

The `Measure` section includes the commands used to perform Event Counting and Event Sampling, with placeholders that will be replaced at runtime while iterating the application measurements

```
[Measure]

perforator_cmd = $PERFORATOR_PATH -V --summary --csv --kernel -e $EVENTS
     -r $FUNCTIONS -o $OUT_PATH -- $EXEC_PATH $EXEC_ARGS
measurement_number = 1
sampling_cmd = perf record -e $EVENTS -F 50000 --strict-freq --
    $EXEC_PATH $EXEC_ARGS
report_cmd = perf report --show-total-period --show-nr-samples --fields=
    overhead,sample,period,sym > $OUT_PATH
```

A.5 Features extraction

The `performance.py` script without arguments iterates over all applications in the `app/source` directory. It is also possible to provide a single `<application-name>` as an argument to extract only its features. All results are inserted under the `results` folder in the output directory configured in `Work_path`: `Work_path/results/<application-name>`. This will contain a `metrics.json` file with the static metrics extracted. Then, one folder for each version of the application considered: `vanilla`, `flatten`, `opaque` will contain ten `event_group_<number>.results.csv` files with the results of the Event Counting for each event group considered. A `sampling` folder would be present with ten more CSV files if the `MeasureSample` option were enabled.

```
python performances.py [app_name]
```

A.6 Merge into dataset

All the output files obtained from one application can be merged into a single `<application-name>.csv` file using the `csv_merge.py` script:

```
python output_analysis/csv_processing/csv_merge.py <input_folder> <
    output_path> [<application-name>]
```

As the `performance.py` script, it can merge all and a single application.

A.7 Train models

The input folder containing the merged applications' CSV files must be passed as an argument to train the machine learning models. The script will internally merge all found files into a single `merged.csv` file, which will be the input dataset for the machine learning models. It only does so if the `merged.csv` file is not yet present in the `<input-folder>`. The results of the model training will be placed in the `output_path`. These outputs include

- a `results.csv` file with the configurations used for the training and the evaluation metrics of the model
- a `predictions.csv` file with all the actual values of the target features and the corresponding predictions
- a `weights_importances.csv` and `weights_importances.png` file with the features importances computed on the model. This is only present when Random Forest is trained

The filename of the outputs also contains some information to quickly see some configuration parameters used during training and to help when dealing with outputs related to many different models.

```
python output_analysis/ML/train_custom.py <input_folder> <output_path>
```

To configure the training parameters, global variables defined at the top of the `train_custom.py` script are used. `Filtering_config` and `Preparation_config` allow customizing which data cleaning procedures to enable or disable, and regulate outlier deviation percentage and validation sample size.

`Training config` allow to choose the machine learning technique to be used, on which obfuscation technique to train the model, the target feature, if the overhead of the feature must be considered or the actual value, and which features of the dataset the model should ignore.

`Random forest config` let customize two hyperparameters used by the Random Forest Regressor.

`Neural network config` let customize the scalers and hyperparameters for the Neural Network Regressor.

```
# Filtering config
FILTER_LONGER_VANILLA =  # Filter out rows where "cpu-clock_flatten" or "cpu-clock_opaque" is lower than "cpu-clock"
FILTER_CONTEXT_SWITCHES_CPU_MIGRATIONS =  # Filter out rows where context switches > 1 and rows where cpu_migrations > 1
FILTER_OUTLIERS =  # Filter out outliers above {OUTLIER_MEDIAN_DEVIATION_PERC} deviation from median
OUTLIER_MEDIAN_DEVIATION_PERC =  # Percentage of deviation from the median over which the row is considered an outlier
BALANCE_DATASET =  # Remove most of the sample of the 2 functions sweeplist and luaC_step that fills most of the dataset

# Preparation config
SPLIT_BY_FUNCTION = # Split the dataset by samples or by functions

VALIDATION_SIZE = # 0.15 = 15% of samples
VALIDATION_SAMPLES_MIN = # 7 = 7% of samples
VALIDATION_SAMPLES_MAX = # 30 = 30% of samples

# Training config
RANDOM_FOREST = # Enable Random Forest training
NEURAL_NETWORK = # Enable Neural Network training
OBFUSCATION_TARGETS = # Models to create ["flatten","opaque"]
TRAIN_COL_NAME= # avg_time-elapsed_ms","cpu-clock","cpu-cycles","ref-cycles","instructions"
PREDICT_OVERHEAD= # consider overhead associated with the target feature, or the actual feature values
EXCLUDE_COLS = ["region","seed","cpu-clock_flatten","ref-cycles_flatten","instructions_flatten","avg_time-elapsed_ms_flatten", "cpu-clock_opaque","ref-cycles_opaque","instructions_opaque","avg_time-elapsed_ms_opaque",] # allows feature selection. always include region, seed and the obfuscated features present on the dataset

# Random forest config
N_THREADS = # Number of threads for training.
N_ESTIMATORS = # Number of estimators (decision-trees) used in the Random Fores
DUMP_MODEL = False # Possibility to dump the resulting model on a file

# Neural network config
NN_TESTING = # Use 10% of the dataset or the full one
INPUT_SCALER = # How to scale input data
OUTPUT_SCALER = # How to scale output data
HIDDEN_LAYERS = [1, 2, 4, 8, 16] # hidden layers in the neural network
NEURONS = [16, 32, 64, 128, 256, 512, 1024] # neurons in the neural network
LEARNING_RATE_INITS = [0.01, 0.05, 0.10]  # initial learning rate of the neural network
```

Appendix B

Developer Manual

This section integrates the implementation details described in Chapter 4 to make it easier for developers to navigate the project structure.

- `performances.py`
  `performances.py` is the coordinator script that handles the metrics extraction workflow. Its configuration instructions can be seen in the User Manual. It coordinates all the operations needed by calling the appropriate functions defined in the other scripts. It also handles Logging the workflow status in `performances.py.exec.log` and handling Exceptions. Errors are escalated to such script using a standardized Exception, ExecError, defined in `utils.py`.

- `utils.py`
  `utils.py` contains most of the logic and utility functions, such as reading configurations and handling folders and paths, as well as source file merging, compilation and obfuscation.

- `metrics.py`
  `metrics.py` handles the collection of static metrics from both the C sources and the assembled executable.

- `benchmark_system_config.py`
  `benchmark_system_config.py` is responsible for implementing Noise Reduction techniques. Its working is explained in detail in Section 4.5

- `measure.py`
  `measure.py` handles dynamic feature extraction. Its two main functions, `collect_dynamic_metrics` and `sample_dynamic_metrics` were explained in detail in Section 4.6

- `output_analysis/csv_processing/csv_merge.py`
  Merge all the .csv files into one file per each program.

- `output_analysis/ML/train_custom.py`
  `train_custom.py` is the script that merges files into the full dataset, performs the data

cleaning procedures selected, and trains the machine learning models according to the selected configuration. The two main functions `trainRandomForestRegressor` and `trainNeuralNetworkRegressor` and the various configuration options are explained in Section 4.7

- `output_analysis/ML/plot_features_importance.py`
  This script takes as input a `...weights_importances.csv` file created by the `train_custom.py` script and outputs a `png` visualizing features importance.

- `output_analysis/ML/create_scatterplot_predicted_vs_actual.py`
  This script takes as input a `...predictions.csv` file created by the `train_custom.py` script and outputs a `html` visualizing an interactive scatterplot of the predicted and actual values.

- `output_analysis/ML/create_histogram_predicted_vs_actual.py`
  This script takes as input a `...predictions.csv` file created by the `train_custom.py` script and outputs a `png` visualizing a histogram of the predicted and actual values.

- `output_analysis/ML/compare_nn_results.py`
  This script takes as input a folder containing multiple `...inverted_results.csv` created by the `train_custom.py` script when Neural Networks are trained and outputs a `rmses.csv` and a `rmses.png` files extracting their `rmse` values from the results. They also create an histogram containing the prediction error values compared between models in `diffs.csv` and `diffs.png`

Appendix C

Events Table

Event name in Perforator (Perf)	**Type**	**Corresponding PERF_TYPE (PERF_COUNT_)**	**Description**	**Group Number**
bus-cycles	Hardware	HW_BUS_ CYCLES	Counts only CPU clock cycles spent on bus transactions	0
cpu-cycles	Hardware	HW_CPU_ CYCLES	Counts CPU clock cycles elapsed	0
ref-cycles	Hardware	HW_REF_CPU_ CYCLES	Counts CPU clock cycles elapsed, not affected by CPU frequency scaling	0
instructions	Hardware	HW_ INSTRUCTIONS	Counts instructions executed by the processor	0
context-switches	Software	SW_CONTEXT_ SWITCHES	Counts context switches occurred	0
cpu-migrations	Software	SW_CPU_ MIGRATIONS	Counts CPU migrations occurred	0
cpu-clock	Software	SW_CPU_CLOCK	Counts Time Stamp Counter clocks elapsed	0
page-faults	Software	SW_PAGE_ FAULTS	Counts page faults occurred: process tries to access a non-existent or non-mapped page in memory	0
branch-instructions	Hardware	HW_BRANCH_ INSTRUCTIONS	Counts branch instructions retired	1
branch-misses	Hardware	HW_BRANCH_ MISSES	Counts mispredicted branch instructions	1

cache-misses	Hardware	HW_CACHE_ MISSES	Counts misses to the cache hierarchy, including reads, writes and prefetches	1
cache-references	Hardware	HW_CACHE_ REFERENCES	Counts accesses to the cache hierarchy, including reads, writes and prefetches	1
alignment-faults	Software	SW_ALIGNMENT_ FAULTS	Counts alignment faults, that occur when the kernel needs to handle an unaligned memory access	1
emulation-faults	Software	SW_EMULATION_ FAULTS	Counts emulation faults, that occur when the kernel traps on an instruction not implemented and needs to emulate it for user space	1
major-faults	Software	SW_PAGE_ FAULTS_MAJ	Counts only page faults that required disk I/O to handle	1
minor-faults	Software	SW_PAGE_ FAULTS_MIN	Counts only page faults that did not require disk I/O to handle	1
bpu-read-accesses (branch-loads)	Cache	HW_CACHE_BPU HW_CACHE_OP_ READ HW_CACHE_ RESULT_ACCESS	Counts read accesses initiated by the Branch Prediction Unit (BPU)	2
bpu-read-misses (branch-load-misses)	Cache	HW_CACHE_BPU HW_CACHE_OP_ READ HW_CACHE_ RESULT_MISS	Counts read misses initiated by the Branch Prediction Unit (BPU)	2
node-read-accesses (node-loads)	Cache	HW_CACHE_ NODE HW_CACHE_OP_ READ HW_CACHE_ RESULT_ACCESS	Counts read accesses to the local memory	2

node-read-misses (node-load-misses)	Cache	HW_CACHE_ NODE HW_CACHE_OP_ READ HW_CACHE_ RESULT_MISS	Counts read misses to the local memory	2
itlb-read-accesses (iTLB-loads)	Cache	HW_CACHE_ ITLB HW_CACHE_OP_ READ HW_CACHE_ RESULT_ACCESS	Counts read accesses to the Instruction Translation Lookaside Buffer (TLB)	3
itlb-read-misses (iTLB-load-misses)	Cache	HW_CACHE_ ITLB HW_CACHE_OP_ READ HW_CACHE_ RESULT_MISS	Counts read misses to the Instruction Translation Lookaside Buffer (TLB)	3
node-write-accesses (node-stores)	Cache	HW_CACHE_ NODE HW_CACHE_OP_ WRITE HW_CACHE_ RESULT_ACCESS	Counts write accesses to the local memory.	3
node-write-misses (node-store-misses)	Cache	HW_CACHE_ NODE HW_CACHE_OP_ WRITE HW_CACHE_ RESULT_MISS	Counts write misses to the local memory.	3
dtlb-read-accesses (dTLB-loads)	Cache	HW_CACHE_ DTLB HW_CACHE_OP_ READ HW_CACHE_ RESULT_ACCESS	Counts read accesses to the Data Translation Lookaside Buffer (TLB)	4
dtlb-read-misses (dTLB-load-misses)	Cache	HW_CACHE_ DTLB HW_CACHE_OP_ READ HW_CACHE_ RESULT_MISS	Counts read misses to the Data Translation Lookaside Buffer (TLB)	4

dtlb-write-accesses (dTLB-stores)	Cache	HW_CACHE_ DTLB HW_CACHE_OP_ WRITE HW_CACHE_ RESULT_ACCESS	Counts write accesses to the Data Translation Lookaside Buffer (TLB)	5
dtlb-write-misses (dTLB-write-misses)	Cache	HW_CACHE_ DTLB HW_CACHE_OP_ WRITE HW_CACHE_ RESULT_MISS	Counts write misses to the Data Translation Lookaside Buffer (TLB)	5
l1d-read-accesses (l1d-loads)	Cache	HW_CACHE_L1D HW_CACHE_OP_ READ HW_CACHE_ RESULT_ACCESS	Counts read accesses to the Level 1 Data Cache	6
l1d-read-misses (l1d-load-misses)	Cache	HW_CACHE_L1D HW_CACHE_OP_ READ HW_CACHE_ RESULT_MISS	Counts read misses to the Level 1 Data Cache	6
l1d-write-accesses (l1d-stores)	Cache	HW_CACHE_L1D HW_CACHE_OP_ WRITE HW_CACHE_ RESULT_ACCESS	Counts write accesses to the Level 1 Data Cache	7
l1i-read-misses (l1i-load-misses)	Cache	HW_CACHE_L1I HW_CACHE_OP_ READ HW_CACHE_ RESULT_MISS	Counts read misses to the Level 1 Instruction Cache	7
ll-read-accesses (LLC-loads)	Cache	HW_CACHE_LL HW_CACHE_OP_ READ HW_CACHE_ RESULT_ACCESS	Counts read accesses to the Last Level Cache (LLC)	8
ll-read-misses (LLC-load-misses)	Cache	HW_CACHE_LL HW_CACHE_OP_ READ HW_CACHE_ RESULT_MISS	Counts read misses to the Last Level Cache (LLC)	8

ll-write-accesses (LLC-stores)	Cache	HW_CACHE_LL HW_CACHE_OP_ WRITE HW_CACHE_ RESULT_ACCESS	Counts write accesses to the Last Level Cache (LLC)	9
ll-write-misses (LLC-write-misses)	Cache	HW_CACHE_LL HW_CACHE_OP_ WRITE HW_CACHE_ RESULT_MISS	Counts write misses to the Last Level Cache (LLC)	9
time-elapsed			Not a perf event, but time elapsed according to the System Clock, retrieved by Perforator.	all

Table C.1: List of events measured

Bibliography

[1] Christian Collberg, Jack Davidson, Roberto Giacobazzi, Yuan Gu, and Amir Herzberg. «Toward Digital Asset Protection». In: *IEEE Intelligent Systems* 26 (Nov. 1, 2011), pp. 8–13. DOI: 10.1109/MIS.2011.106.

[2] BSA — THE SOFTWARE ALLIANCE. *Bsa global software survey*. June 1, 2018. URL: https://gss.bsa.org/wp-content/uploads/2018/05/2018_BSA_GSS_Report_en.pdf.

[3] *Executive Summary Online Copyright Infringement Tracker Survey (12th Wave)*. GOV.UK. URL: https://www.gov.uk/government/publications/online-copyright-infringement-tracker-survey-12th-wave/executive-summary-online-copyright-infringement-tracker-survey-12th-wave.

[4] Stefano Alberto. *Towards the Prediction of Performance Degradation of Obfuscated Code*.

[5] Antonio Licursi. *Data set generation for performance overhead prediction of obfuscated code*. 2021.

[6] *Introduction*. URL: https://tigress.wtf/introduction.html.

[7] Andy Liaw and Matthew Wiener. «Classification and Regression by randomForest». In: 2 (2002).

[8] Fionn Murtagh. «Multilayer Perceptrons for Classification and Regression». In: *Neurocomputing* 2.5 (July 1, 1991), pp. 183–197. ISSN: 0925-2312. DOI: 10.1016/0925-2312(91)90023-5. URL: https://www.sciencedirect.com/science/article/pii/0925231291900235.

[9] Denis Bakhvalov. *Denis Bakhvalov - Performance Analysis and Tuning on Modern CPUs.Pdf*. 2020.

[10] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Software Security Series. Upper Saddle River, NJ: Addison-Wesley, 2010. 748 pp. ISBN: 978-0-321-54925-9.

[11] *Perf Wiki*. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[12] Jessica Rudd and Herman Ray. «An Empirical Study of Downstream Analysis Effects of Model Pre-Processing Choices». In: *Open Journal of Statistics* 10 (Jan. 1, 2020), pp. 735–809. DOI: 10.4236/ojs.2020.105046.

[13] Glosser.ca. *English: Artificial Neural Network with Layer Coloring*. URL: https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg.

[14] Vagelis Plevris, German Solorzano, Nikolaos Bakas, and Mohamed Ben Seghier. *Investigation of Performance Metrics in Regression Analysis and Machine Learning-Based Prediction Models*. June 1, 2022. DOI: 10.23967/eccomas.2022.155.

[15] *Developer Options (Using the GNU Compiler Collection (GCC))*. URL: https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html.

[16] Konrad Weihmann. *Priv-Kweihmann/Multimetric*. Mar. 20, 2024. URL: https://github.com/priv-kweihmann/multimetric.

[17] Dolores R Wallace, Arthur H Watson, and Thomas J McCabe. *Structured Testing : A Testing Methodology Using the Cyclomatic Complexity Metric*. NIST SP 500-235. Gaithersburg, MD: National Institute of Standards and Technology, 1996, NIST SP 500–235. DOI: 10.6028/NIST.SP.500-235. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-235.pdf.

[18] T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. «Software Complexity Analysis Using Halstead Metrics». In: *2017 International Conference on Trends in Electronics and Informatics (ICEI)*. 2017 International Conference on Trends in Electronics and Informatics (ICEI). May 2017, pp. 1109–1113. DOI: 10.1109/ICOEI.2017.8300883. URL: https://ieeexplore.ieee.org/document/8300883.

[19] *TIOBE Quality Indicator*. TIOBE. URL: https://www.tiobe.com/quality-models/tqi/.

[20] *Radareorg/Radare2*. radare org. URL: https://github.com/radareorg/radare2.

[21] URL: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[22] *Ia-32-Ia-64-Benchmark-Code-Execution-Paper.Pdf*. 2010.

[23] *Linux Manpages Online - Man.Cx Manual Pages*. URL: https://man.cx/cset-shield.

[24] *Linux Manpages Online - Man.Cx Manual Pages*. URL: https://man.cx/perf_event_open.

[25] *Zyedidia/Perforator: Record "Perf" Performance Metrics for Individual Functions/Regions of an ELF Binary*. URL: https://github.com/zyedidia/perforator.

[26] *Feature Importances with a Forest of Trees*. scikit-learn. URL: https://scikit-learn/stable/auto_examples/ensemble/plot_forest_importances.html.

[27] *Sklearn.Ensemble.RandomForestRegressor*. scikit-learn. URL: https://scikit-learn/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html.

[28] *Sklearn.Neural_network.MLPRegressor*. scikit-learn. URL: https://scikit-learn/stable/modules/generated/sklearn.neural_network.MLPRegressor.html.

[29] *Regression Using a Scikit MLPRegressor Neural Network* -. Visual Studio Magazine. URL: https://visualstudiomagazine.com/Articles/2023/05/01/regression-scikit.aspx.

[30] *Compare the Effect of Different Scalers on Data with Outliers*. scikit-learn. URL: https://scikit-learn/stable/auto_examples/preprocessing/plot_all_scaling.html.

[31] *Sklearn.Preprocessing.StandardScaler*. scikit-learn. URL: https://scikit-learn/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

[32] *Sklearn.Preprocessing.RobustScaler*. scikit-learn. URL: https://scikit-learn/stable/modules/generated/sklearn.preprocessing.RobustScaler.html.

[33] Jason Brownlee. *How to Use Data Scaling Improve Deep Learning Model Stability and Performance*. MachineLearningMastery.com. Feb. 3, 2019. URL: https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/.

[34] Sebastian Banescu. «Characterizing the Strength of Software Obfuscation Against Automated Attacks». July 18, 2017. DOI: 10.13140/RG.2.2.36593.79202.