This article has been accepted for publication in IEEE Transactions on Emerging Topics in Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TETC.2024.3439884

Enabling Obfuscation Detection in Binary Software through eXplainable AI

Claudia Greco*, Michele Ianni*, Antonella Guzzo, Giancarlo Fortino, *Fellow, IEEE*
Department of Computer Science, Modeling, Electronic and System Engineering (DIMES),
University of Calabria, Italy
Email:[claudia.greco, michele.ianni, antonella.guzzo, giancarlo.fortino]@unical.it

***Abstract*—Binary obfuscation techniques are commonly employed to protect code against reverse engineering and piracy. Unfortunately, besides being used for legitimate purposes, virus writers also resort to obfuscation to evade antivirus detection mechanisms based on signature scanning. Consequently, the detection of obfuscated code in executables may be a precious resource to prevent the execution of malicious programs. Detecting obfuscation is a task fraught with difficulties owing to the wide range of possible obfuscation transformations and the indistinguishability of obfuscated code. In this paper, we venture into the not-so-explored world of obfuscation detection, gaining a deeper comprehension of what happens - from a statistical perspective - to a binary program when obfuscation transformations are applied to it. We accomplish this goal by leveraging eXplainable Artificial Intelligence, which allows us to discern the altered features from the invariant ones, which in turn can then be used for obfuscation-resilient malware detection. The present study has been carried out utilizing diverse datasets, not only to examine the detection of obfuscation but also to classify the specific obfuscating transformations employed. The investigation encompasses binaries compiled for various architectures, and we propose an effective methodology for identifying both the existence of obfuscation and isolating invariant patterns that can facilitate the creation of obfuscation-resistant signatures for antivirus detection.**

***Index Terms*—Obfuscation detection, Static analysis, XAI.**

I. Introduction

Over time, software has become a highly valuable resource that necessitates to be preserved from a plethora of possible attacks, including piracy, which poses a major concern for the software industry. In order to protect software from code stealing, tampering, or cracking, it is necessary to implement countermeasures aimed at ensuring code confidentiality and preventing reverse engineering. Software protection is an ongoing area of research, with new solutions being constantly sought [1], whose actual effectiveness is still a topic of debate. In spite of that, code obfuscation stands as the most widely employed approach, although it only ensures temporary safeguarding [2] and does not offer a conclusive remedy to the issue of protecting code. Obfuscation refers to a collection of techniques that aim to modify the appearance of a program while retaining its original behaviour. The application of obfuscation strategies results in a substantial increase in the complexity of reverse engineering and manipulation, thus serving as an effective countermeasure for safeguarding critical code. Actually, code obfuscation is a technique that is not only used in the realm of software protection, but is also widely adopted by malware authors, who employ it to hide malicious tasks within programs and make it appear as legitimate code. In fact, as the code is transformed into a different form from its original, it is not recognized by traditional antivirus software that rely on a signature scanning mechanism: they scan the target executable for an invariant code pattern, known as a signature. This pattern comprises a sequence of bytes representing code or data and allows identifying the code as malicious. Through the use of code obfuscation, malware writers are able to transform their code into new variants that still perform their malicious function, but do not exhibit the code pattern that identifies them, allowing evasion from signature-based detection by antivirus systems. In this domain, the task of detecting obfuscated code into software becomes highly critical, as it constitutes a first line of defense against malicious code. However, performing obfuscation detection is a challenging task, especially in a context involving a large number of potentially malicious programs that are advised not to be executed. On closer inspection, the task of obfuscation detection emerges as a significantly important activity, not only for countering malicious actors but also as a process intended to evaluate the efficiency and robustness of tools used in the field of software protection.

This paper aims to offer a fresh viewpoint to obfuscation detection by conducting a static analysis on various features of executables. First of all, we provide an approach, based on machine learning, for allowing the detection of obfuscated code, along with the distinction of the type of transformations applied, which is the first step towards reverse engineering and de-obfuscation. Then, we employ eXplainable Artificial Intelligence (XAI) to investigate the effects that obfuscation transformations have on binary programs and their static properties, whose variation can be indicative of the presence of obfuscated code. Specifically, through this study, we aim to provide a key interpretation of the static features of binaries. In order to achieve this goal, we conduct an analysis that leads us to identify the features that vary the most when transformations are applied and those that vary very little or not at all. This latter aspect is also of utmost interest, as these features can be of great value for generating signatures intended for obfuscation-resistant malware detection. Our approach involves an automated static extraction of properties from binaries compiled using different compilers, op-

*These authors contributed equally to this work.

timization levels, and obfuscation transformations, which are subsequently fed into ML and XAI algorithms. The resulting dataset forms a crucial component of our study. Each entry in this dataset represents a compiled binary, accompanied by a label indicating whether a binary has been subjected to obfuscation and, if so, the specific type of obfuscation applied. The utilization of binaries compiled for diverse architectures is a crucial aspect in this study. The prevalence of malware in contemporary times often involves the compilation of original source code for multiple platforms. Consequently, detecting such malware becomes arduous due to the ineffectiveness of signatures designed for one architecture to identify the same source code compiled for another architecture, mainly due to variations in opcodes. The distinct instruction sets associated with different architectures introduce additional complexities in the analysis process. Notably, the field of obfuscation detection remains relatively underexplored within the existing literature and, to the best of our knowledge, this paper represents the pioneering effort in its category, forging a novel trajectory for obfuscation detection analysis. This paper extends and redefines the approach proposed in [3]. In comparison to the referenced paper, our contributions encompass substantial novelties. These include but are not limited to the identification of obfuscation types, detection of multiple obfuscations applied to a singular binary, exploration across various architectures, and a substantial expansion of the utilized dataset.

In summary, we provide the following significant contributions:

- A new framework for the static extraction of essential features from binary files, enhancing the ability to analyze and detect obfuscation. We also built a comprehensive dataset comprising features extracted from open-source code by applying our framework.
- An advanced machine learning-based approach to obfuscation detection that not only identifies the presence of obfuscation but also excels in distinguishing among distinct types of obfuscation.
- An extensive analysis of static features that are mostly influenced when specific obfuscating transformations are applied; consequently, we also identify the features least influenced by obfuscating transformations.

The remainder of the paper is structured as follows: in Section II we provide basic concepts of obfuscation and specific obfuscating transformations, uses of code obfuscation in malware writing, and explainability through XAI. In Section III we shortly review the state of the art regarding the employment of ML models to address the obfuscation detection problem. Section IV explains our approach for detecting obfuscation using automated feature extraction. We present the results and understandings obtained through our proposal in Section V and, finally, in Section VI, we derive conclusions from our discoveries and delve into discussions about future work.

II. Background

A. Code Obfuscation

Code obfuscation refers to a software protection technique that involves applying a series of transformations to code with the goal of altering its syntax while preserving its semantics. On the resulting code, it becomes more challenging to perform reverse engineering, extract information from it, modify its behavior, or, more generally, analyze it. The obfuscation process is carried out by an *obfuscator* that applies a sequence of transformations $T = \{t_1, t_2, ..., t_n\}$ to a program P, with the goal of transforming it into a new program P'. The transformations in T preserve the code semantics: P' still behaves as P, meaning that the two programs are semantically equivalent, but they present different syntax, with P' usually being much harder to analyze with respect to P.

According to Collberg's taxonomy [4], code transformations are categorized into three primary classes, encompassing different types of transformations:

- *Layout Transformations*: Such transformations alter the program's layout structure, for instance by modifying identifiers or eliminating debugging information. The majority of layout obfuscation is non-reversible, involving actions such as substituting identifiers with arbitrary symbols and eliminating comments, redundant methods, and debugging information. While layout obfuscation may not completely prevent reverse engineering via code inspection, at least it increases the cost of the operation.
- *Control Transformations*: The purpose of these transformations is to make it difficult for an adversary to track a program's control flow and generate analysis structures, such as control flow graphs. Control transformations include inserting bogus control flow instructions, obfuscating the targets of branch instructions, or removing all structured programming constructs to flatten the program.
- *Data Transformations*: The data structures present in the source application are transformed by disguising them into an intricate form that is difficult to analyze. A data transformation can be represented as a function, denoted as $E()$, that converts a variable, V, into a representation that is obscured. All feasible values for V, along with every legitimate action that the program can perform on V, must also be translated into this new representation.

Unfortunately, the employment of obfuscation transformations comes at a cost, as the resulting program typically increases in size, execution time and memory footprint. To minimize costs, the application of obfuscation is often restricted to specific parts of code or data related to the information requiring protection.

B. Code Obfuscation uses for Malware Writing

In addition to being an important tool for software protection, obfuscation techniques are widely used in the development of malware since they allow to bypass traditional signature scanning-based antivirus systems. In fact, despite being an active research area with new solutions proposed [5], [6], malware detection is still primarily addressed through

signature scanning-based techniques [7]. A signature can be defined as a fixed and unalterable sequence of bytes, usually obtained from either the code or raw content of an application, which serves the essential purpose of providing a unique identification for a particular malware.

More formally, as proposed by Bonfante *et al.* [8], given the set of all programs P, the set of malware M, with $M \subset P$, the set of signatures $\mathbb{S}$, and a function called *detector*, we define (D) as $D : P \rightarrow \{0, 1\}$. Let p be a program. We say that p is a detected malware if a signature $m \in \mathbb{S}$ such that $D(p, m) = 1$ exists and this happens if and only if m is a sequence extracted from the program p. In response to this detection strategy, malware authors have developed *metamorphic* viruses, which utilize self-modification techniques while propagating in order to create new variants that exhibit different forms while preserving the same functionality. These variants aim to evade signature-based detection methods by changing their appearance, thereby increasing their stealthiness and reducing their chances of being detected [9]. The purpose of this intricate task is to modify the unchanging patterns utilized as malware signatures. This is accomplished by applying a diverse range of code transformations, including, but not restricted to: permutation, garbage insertion, expansion, shrinking, register swaps, encryption, and other similar techniques [4]. The use of signatures for malware detection, however, has its pros and cons. On the one hand, if we compare this strategy to dynamic analysis-based methodologies, we can easily notice that signature detection is faster in terms of scanning time, produces a lower number of false positives and, since the malware is not executed, it avoids accidental system infections; on the other hand, however, it is able to detect only known malware, whose signature has already been defined. Malware developers frequently use runtime packers, which are self-extracting archives that unpack in memory upon execution, along with code obfuscation techniques to hinder reverse engineer analysts from analyzing and make it challenging for signature scanners to detect malware. Bat *et al.* [10] reported that the utilization of runtime packers is observed in over 80% of malicious software, while nearly half of the new malware instances are produced by repacking the already existing malware. Several studies have shown that obfuscation is a widely-used strategy to evade signature-based malware detection, as evidenced by works such as [11]–[14]. Obfuscation is a long-standing technique and, aside from packing, encryption can be considered the first form of obfuscation technique seen in the wild [9], [15]–[18]

Malware encryption is a method that involves encrypting the program's body and including a decryptor that decrypts the code during runtime. The strength of this technique lies in the fact that the key used for encryption changes with every infection, making each new variant unique. Among the first malwares to use this technique, we find Cascade, Win95/Mad and Win95/Zombie [19], and some of them, such as Win32/Coke, also implemented multiple levels of encryption. The weakness of this approach lies in the fact that, unlike the program body, the decryptor typically remains unchanged, thus representing a constant piece of code from which signatures can be extracted. To circumvent this limitation, malware authors introduced a new type of malware called *polymorphic viruses* [13], which employ different schemes to alter the decryptor as well. Examples of early malware employing a 32-bit polymorphic engine were Win95/Marb.urg and Win95/HPS. Although polymorphic viruses are quite effective in evading signature-based antivirus scanners, they can be easily detected with dynamic approaches, since the body of the virus will be unencrypted in memory at runtime and using sandboxing techniques for controlled malware execution [16], [17] can lead to a convenient analysis and signature identification. Another significant advancement in the evolution of malware was marked by the emergence of *metamorphic viruses* [11], [14], [17], [18], [20], [21], which are capable of altering their code structure with each infection. However, this mutation is purely syntactic in nature, leaving the malware's functional behavior unchanged.

The techniques applied in order to achieve this goal are numerous, and many of them were already used in polymorphic viruses:

- *Register swapping* consists in altering the registers used in various instructions while generating a new version of a virus. This technique, historically attributed to the malware writer Vecna for his virus Win95/Regswap [18], can be easily dismantled by leveraging wildcard-based signatures and regular expression scans.
- *Instruction substitution* refers to replacing certain instructions or groups of instructions with others that have the same functionality but are syntactically different [21].
- *Garbage instructions insertion* involves adding instructions that are not necessary for the program's execution flow, with the goal of varying the malware body [18], [21], [22]. These instructions can be single operations or sequences that do not affect the program's state, and may even be located in areas that will never be executed ("dead-code").
- *Transposition* involves various techniques for reordering instructions while preserving the original flow of the program [20]. One method is to randomly reorder some instructions and then use unconditional jumps to reconstruct the original flow. Another approach is to isolate independent groups of instructions and modify their order, eliminating the need for unconditional jumps. However, finding independent groups of instructions can be challenging, so developers often resort to simpler techniques, such as reordering subroutines within the malware. This approach was used by the Win32/Ghost malware and can result in up to $n!$ different variants, where n is the number of subroutines.
- *Code integration*, one of the most sophisticated techniques used to obfuscate code was first employed by the virus writer Zombie in Win95/Zmist (Zombie Mistfall). It involves decompiling the program to be infected, inserting malware code in between, and then reassembling the original program and malware into a single executable.

C. Explainable Artificial Intelligence

Artificial intelligence (AI) models have long been regarded as "black boxes" by humans, indicating that they offer re-

sponses without providing any insight into how their decisions are formulated. This has led to the emergence of a subfield of AI known as eXplainable AI (XAI), which focuses on elucidating the mechanisms underlying AI system predictions. Among XAI techniques are included rule-based systems, decision trees, and model-agnostic methods such as LIME (Local Interpretable Model-agnostic Explanations) and SHAP (SHapley Additive exPlanations). These methods serve the purpose of providing explanations for decisions made on individual instances or offer a comprehensive overview that explains the behavior of the generated model. A generally acknowledged classification of XAI methods is as follows:

- *Model Agnostic/Specific*: Interpretation techniques that are specific to a particular model are referred to as model-specific, while those that are not are known as model-agnostic. Although agnostic approaches can enhance interpretability of any machine learning model, they may be precluded from having access to internal model data such as weights and structural information.
- *Intrinsic/Extrinsic*: Models can be inherently interpretable, or interpretation techniques may be necessary post-training to achieve interpretability. Decision trees and other transparent models are intrinsic models, whereas extrinsic models require the use of interpretation techniques after training.
- *Local/Global*: Local interpretation techniques capture the behavior of a single instance, while global interpretation techniques depict the behavior of the entire dataset.

D. Model Interpretability using SHAP

The SHAP methodology is deeply rooted in cooperative game theory, specifically focusing on the computation of the Shapley value [23]. This value serves as a metric to gauge the influence exerted by an individual player in the formation of coalitions. Within the realm of game theory, a coalitional game comprises a cohort of N players and a characteristic function, denoted as v, mapping player subsets, $S \subseteq 1, 2, \cdots, N$, to a real value, $v(S)$ — representing the collective payoff achieved by the players in the given subset when collaborating. The Shapley value, in turn, is derived through a weighted average computation of a player's marginal contributions across all conceivable player subsets. Here, the marginal contribution, $\Delta v(i, S)$, ascribed to player i within coalition S, represents the additional value brought about by the inclusion of player i in the coalition. In the context of XAI, consider a model prediction m constructed upon a set of d features, denoted as $m(f_1, f_2, \cdots, f_d)$. Consequently, features 1 through d can be analogously perceived as players in a game. Here, the payoff, $v(S)$, with $S \subseteq f_1, f_2, \cdots, f_d$, takes the form of a scalar prediction computed from a subset of feature values. The Shapley value computation, denoted as ϕ_i, for each feature, then mirrors the contribution of that feature to the overall model prediction:

$$\phi i = \sum S \subseteq F \setminus i \frac{|S|!(|F| - |S| - 1)}{|F|!} \Delta v(i, S) \qquad (1)$$

Here, F represents the set of all features, while S is a subset of F. The term $\Delta v(i, S)$ delineates the marginal contribution attributed to feature i — essentially, the difference between the model prediction when including feature i and when excluding it.

The precise computation of the Shapley value requires an evaluation of the characteristic function a factorial number of times, resulting in a computational complexity denoted by $O(2^n)$, where n is the number of features [23]. It is easy to notice that training an ML classifier on all the subsets to compute the exact Shapley value is computationally infeasible. To overcome this problem, several approximation techniques have been introduced in recent years to calculate the Shapley value efficiently. Such techniques vary in the assumptions and computational methods they apply for approximating $\Delta v(i, S)$ in Equation 1, with the expected decrease in the prediction variance if we remove the input variable x from the subset S of the input features $E[m(x)|x] - E[m(S - x)|S - x]$.

III. Related Work

Despite the widespread use of obfuscation in numerous application scenarios, most of the studies in the scientific literature focus on its detection in Javascript and Android programs, with very limited attention on the binary domain. Several papers explore the application of machine learning methods for detecting obfuscation [24]–[26].

Salem *et al.* [24] introduced Oedipus, a machine learning-based framework designed to perform metadata recovery attacks on obfuscated C programs. The framework leverages Decision Tree and Naive Bayes classifiers to detect six specific obfuscation transformations. To evaluate Oedipus, the authors employed the `Tigress` obfuscator to generate a dataset of 11,075 obfuscated programs, all compiled for the x86 architecture. They employed Term Frequency Inverse Document Frequency (TF-IDF) features within the disassembly files. The primary goal of Oedipus is not to distinguish between obfuscated and non-obfuscated programs. Instead, it aims to identify specific obfuscation transformations, enabling the recognition of particular patterns and techniques used by an obfuscator when applying one of the six considered transformations. Moreover, the study considers only a single possible layer of obfuscation, which represents a limitation, as multiple transformations are frequently applied to programs in practical scenarios. Shirazi *et al.* [25] investigated the application of semantic reasoning in combination with ensemble learning for detecting obfuscation transformations at the basic-block level. The authors employed the `Tigress` and `O-LLVM` obfuscators to generate obfuscated versions of code, producing programs with both single and multiple layers of obfuscation. However, the study lacks precise information regarding the dataset size, which is stated to be between 1,000 and 5,000 samples, as well as the specific features chosen and extracted from the binaries. Additionally, focusing on basic-block level granularity may limit the detection of various transformations, such as those that impact the overall layout of a program. Jiang *et al.* [26] developed a hybrid neural network model, for detecting obfuscation in x86 assembly code by combining Graph

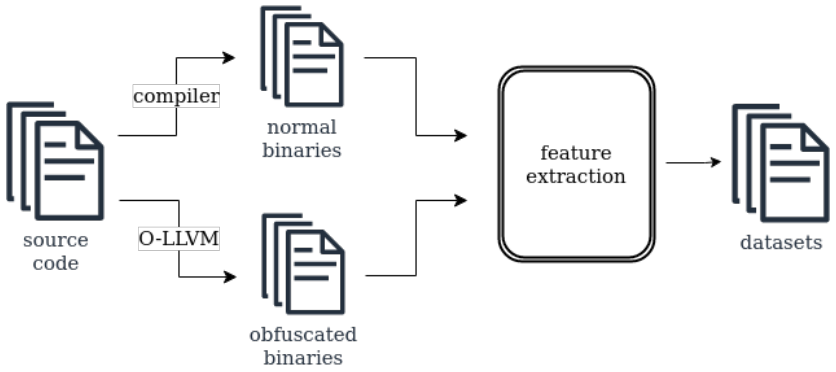

Fig. 1. *Dataset generation*

Convolutional Network (GCN) and Long Short-Term Memory (LSTM) techniques. The authors generated a comprehensive dataset consisting in a total of 43,381 binaries compiled for the x86 architecture, where the obfuscated binaries were produced using `O-LLVM`. The authors selected 27 features for their analysis, most of which are related to the count of specific instruction types (e.g., number of arithmetic instructions, number of logic instructions, and so forth). However, this choice could be a limitation, as these features are absolute values that may lack meaningful significance when comparing programs of different sizes.

Some attempts have also been made to investigate alternative approaches that do not rely on machine learning for detecting obfuscation [27]–[29]. Smith *et al.* [27] presented a static approach, called REDIR, specifically tailored for obfuscated anti-debugging methods, which utilizes a rule based engine leveraging intermediate representation (IR). A tool called DynODet is proposed in [28]. It is a binary instrumentation-based tool designed to detect dynamic obfuscation. Specifically, DynODet is able to detect six different types of obfuscation transformations that were observed being highly represented in a set of binary programs. Lastly, Treadwell *et al.* [29] developed a heuristic approach for detecting malicious obfuscated PE binaries in Windows systems that leverages static features.

IV. Proposal

This paper introduces a machine learning-driven method for detecting obfuscation by relying on XAI to identify the impact of specific transformations on features related to program properties. To conduct our investigation, we constructed diverse datasets by extracting 19 features from a total of 102,146 different binaries (including both non-obfuscated and obfuscated binaries). To the best of our knowledge, our study represents the largest-scale investigation of obfuscation detection involving such a substantial number of binaries.

The methodology employed for creating our datasets is summarized in Figure 1.

Initially, we compiled an extensive collection of source code from open-source repositories of tools and libraries. Notable projects considered for this purpose include `binutils`, `coreutils`, `inetutils`, `gzip`, `tar`, and `sharutils`, among others. To compile the code, we utilized various versions of the GNU Compiler Collection [*] (`GCC`): 4.9.4, 5.5.0, 6.4.0, 7.3.0, and 8.2.0, along with versions 4.0, 5.0, 6.0, and 7.0 of `Clang` [*]. It is important to clarify that our focus is not on the specific version but rather on the inherent variability introduced by different compilers and optimization levels. The selected versions were chosen to represent a range of compiler releases over time. We also considered compiler optimization, as it shares similarities with obfuscation by transforming the original code to enhance performance and/or reduce code size, and may lead to increased compilation time and potentially affect the program's debuggability [*]. We employed five distinct optimization levels, from `O0` to `O3`, in addition to the code size optimization option, `Os`. The inclusion of multiple optimization levels in our study was motivated by the objective of strengthening the classification process. By considering various optimization levels, we aimed to enhance the classification model's ability to differentiate obfuscating transformations from optimization techniques.

Meanwhile, to generate the obfuscated binaries, we compiled the source code using the `O-LLVM` (`Obfuscator-LLVM`) tool [30]. We applied three different obfuscating transformations: *Control-Flow Flattening*, *Bogus Control Flow* and *Instruction Substitution*. The *Control-Flow Flattening* transformation, as the name suggests, completely rearranges the flow of a program [31], [32]. This process involves simplifying the connections between different parts of the program. Both conditional (decisions) and unconditional (straightforward) paths are rerouted through a central hub called the dispatcher node. This node uses a fabricated variable to decide where the program should go next. The value of this variable changes after each program section, known as a basic block, is completed. The *Bogus Control Flow* transformation modifies a program's function call graph by inserting extra steps before the existing ones. These new steps include what is called an opaque predicate, which is a condition whose outcome is hard to predict, followed by a decision to return to the original steps. Additionally, the original steps are copied, and random unnecessary instructions are added to the copied version.

[*] https://gcc.gnu.org/
[*] https://clang.llvm.org/
[*] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

This method changes how the program moves through the opaque predicate and decision, ultimately creating a different function call graph. Adding pointless instructions in the copied steps also makes the code more confusing and harder to analyze. *Instruction Substitution* involves replacing original instructions with functionally equivalent but syntactically different instructions. By substituting instructions, the obfuscator alters the program's control flow and data flow, making it harder for reverse engineers to understand its logic and purpose. The goal of instruction substitution is to introduce confusion and increase the complexity of the binary code, thereby impeding static analysis and making the reverse engineering process more challenging.

We developed a framework enabling the feature extraction of a set of 19 features, as presented in Table I, which encompasses significant properties of the program under analysis, including the class label. The feature extraction mechanism was implemented using Python and leveraged the Python APIs of `radare2`*, specifically the `r2pipe` module. We emphasize that all the chosen features pertain to functions since the obfuscating transformations that we considered can be implemented at the function level. The feature extraction process employed in this study is entirely static. This method was necessary to reproduce a realistic scenario wherein running obfuscated code might lead to running malicious code. Thus, we adopted a static extraction procedure to integrate our framework into a detection system, enabling the generation of alerts when obfuscating transformations are detected. Furthermore, our static methodology offers additional advantages. Static analysis methods are faster compared to dynamic analysis. They are also capable of examining all possible execution paths, not only those executed in practice. Hence, static analysis can detect obfuscated code fragments that may only manifest in rare circumstances during execution. This aspect of static analysis is particularly valuable in the presence of obfuscated routines, which may only execute if stimulated with specific inputs or when a precise event occurs. Another notable advantage of our static analysis approach is its applicability to binaries compiled for different operating systems and architectures. Since we do not require binary execution, we avoid dealing with complicated and costly emulation, enabling us to extract features of interest from every binary, regardless of its architecture or format. This feature is particularly significant for our proposal, as we deal with many binaries compiled for different architectures, demonstrating the effectiveness of our approach in scenarios such as the Internet of Things (IoT), where various architectures are in use. The binaries are compiled for the following architectures: `x86_32`, `x86_64`, `arm_32` (little endian), `arm_64` (little endian), `mips_32` (little endian), `mips_64` (little endian), `mipseb_32` (big endian), and `mipseb_64` (big endian). Considering different architectures holds significant importance from our perspective. Not only is this issue not extensively explored in the existing literature, but it is also essential to consider that a substantial portion of malicious software targeting the IoT consists of repurposed versions of malware originally designed for other architectures. Thus, developing novel approaches for detecting malicious code that are independent of the underlying architecture is crucial. This way, malicious behavior can be readily identified even if the code is compiled for different architectures or has undergone modifications through various compiler optimizations or obfuscating transformations.

From the generated list of binaries, we created six distinct datasets for our experiments:

- The first three datasets, named `fla`, `bcf`, and `sub`, are utilized for a binary classification task. Each dataset comprises both non-obfuscated programs and obfuscated ones. The `fla` dataset involves the control flow flattening transformation, the `bcf` dataset includes the bogus control flow transformation, and the `sub` dataset incorporates the instruction substitution transformation.
- The fourth dataset, named `binary mixed`, encompasses all the non-obfuscated binaries along with their respective obfuscated versions, each transformed using a single obfuscating technique. This dataset is also used for binary classification, where the binaries are labeled as either obfuscated or non-obfuscated.
- The fifth dataset, named `multi partial`, involves a multi-class classification task, with four different class labels (obfuscated or a single applied transformation). This dataset includes all non-obfuscated binaries and obfuscated binaries generated by applying a single obfuscating transformation.
- The last dataset, named `multi all`, also consists of a multi-class classification task. In this case, we augment the fifth dataset with additional obfuscated binaries obtained by applying three different obfuscating transformations to the same binary. The purpose of experiments involving this dataset is to determine if we can detect the presence of obfuscation even when multiple transformations are applied to a binary. Additionally, we aim to identify features that can effectively detect the usage of multiple transformations.

Table II presents the instruction categories extracted from the binaries, which are used to calculate the percentages shown in Table I. It should be noted that the categories listed do not include specific instructions but rather represent instruction types. These types are abstractions created by `radare2` and represent sets of instructions. As an example, the `je` and `jle X86` instructions are classified under the `cjmp` instruction type, despite being distinct instructions. These abstractions enable us to manage extensive lists of different instructions across various architectures by organizing them into feasible categories that are divided by high-level behavior.

The datasets underwent preprocessing before being utilized in a *XGBoost* classifier incorporating *k-fold cross-validation*. It is crucial to note that multiple samples, generated from each program used in the study with variations in optimization levels, compilers, and obfuscation techniques, were considered. These samples are not independent; binaries produced at the same optimization level by different compilers, such as `GCC` and `Clang`, are likely to exhibit significant similarities. To address this, special care was taken during the train-test split

*https://rada.re/

This article has been accepted for publication in IEEE Transactions on Emerging Topics in Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TETC.2024.3439884

TABLE I
EXTRACTED FEATURES

Feature	Meaning
f_cyc_complex	Cyclomatic complexity
f_cycle_cost	Function cycles cost
f_loop_count	Loop count
f_n_bb	Number of basic blocks
f_n_locals	Number of local variables
f_arit	Percentage of arithmetic instructions
f_logic	Percentage of logic instructions
f_control_flow	Percentage of control flow instructions (e.g. `jmp`)
f_stack	Percentage of stack operations (e.g. `push` and `pop`)
f_memory	Percentage of memory access operations
f_program_flow	Percentage of program flow instructions (e.g. `call` and `ret`)
n_func	Number of functions
n_imp	Number of imported symbols
n_string	Number of strings
n_symbols	Number of defined symbols
f_n_xrefs	Number of cross-references to current function
f_n_instr	Number of instructions
f_edges	Number of edges in the Control Flow Graph

TABLE II
INSTRUCTIONS CATEGORIES

Category	Instructions Types
Arithmetic Operations	`add`, `div`, `mod`, `mul`, `sub`, `abs`
Logic Operations	`and`, `not`, `or`, `xor`, `rol`, `ror`, `sal`, `sar`, `shl`, `shr`, `cmp`, `acmp`
Control Flow	`cjmp`, `mjmp`, `jmp`, `ucjmp`, `ujmp`, `rjmp`, `ijmp`, `irjmp`, `mcjmp`, `rcjmp`, `cmov`, `case`, `switch`, `trap`
Stack Operations	`pop`, `push`, `upush`, `rpush`
Memory Access	`mov`, `lea`, `load`, `store`
Program Flow Control	`call`, `ccall`, `leave`, `ret`, `cret`, `ucall`, `rcall`, `icall`, `ircall`, `ucccall`

process. Specifically, we ensured that samples generated from the same program never appeared in both the training and testing datasets. This precautionary measure prevents the inflation of accuracy results that could arise if identical samples were present in both sets, thereby enhancing the reliability of our evaluation. The selection of XGBoost was motivated by its advantageous qualities, including its capacity to generate interpretable models. This is due to the algorithm's reliance on decision trees, which enable visualization and analysis, facilitating the comprehension of the associations between features and predictions. Consequently, it becomes more convenient to offer explanations for individual predictions. Furthermore, XGBoost has been integrated with diverse XAI algorithms, including feature importance scores and partial dependence plots, which provide additional knowledge about the model's classification process.

In our analysis, we employ the SHAP methodology, as introduced by Lundberg and Lee [33]. This game-theoretic approach is chosen for its numerous advantages over other prevalent methods, making it a preferred option for explaining the outcomes of our machine learning model. A pivotal strength of SHAP lies in its extrinsic nature, rendering it suitable for ex-post analysis. Notably, its model-agnostic characteristic enhances its versatility, accommodating a diverse array of models, including but not limited to XGBoost. This stands in contrast to alternative techniques such as LIME [34] and Anchors [35], which are localized in their applicability and encounter constraints when confronted with varied model architectures. Additionally, while Gradient-weighted Class Activation Mapping (GRAD-CAM) [36] and other gradient-based methodologies share an extrinsic nature, they are inherently model-specific, which potentially limits their applicability across the spectrum of models. The adoption of SHAP serves a dual purpose: besides supplying a localized interpretation of a prediction, it also offers a global interpretation. This comprehensive approach ensures an understanding of the predictive process, encapsulating the influence of all features integrated into the model. Such a dual-tiered interpretation is especially crucial for enhancing interpretability and understanding the rationale behind the prediction.

In this paper, we rely on the use of a TreeExplainer [37], a powerful model for the computation of SHAP values. The TreeExplainer algorithm is usually employed when additive tree-based models, such as random forests and gradient boosting machines, are used for the classification. The key idea underlying the algorithm is to compute an estimate of the expected value for the model's output, denoted as $f(x)$, given a specific subset of features, represented as x_S, $E[f(x)|x_S]$. To achieve this estimate, the algorithm looks at how many training samples with feature values matching x_S end up in each leaf node of the decision tree. By examining the distribution of these matching samples across the leaf nodes, the algorithm gains information about the impact of the features on the model's predictions. The proportion of matching samples in each leaf node provides insight into how much each leaf node contributes to the overall prediction for the specific feature subset x_S. One of the key advantages of the TreeExplainer method for computing SHAP values is its ability to significantly reduce computational complexity compared to exact SHAP value computation. The reduction in computational complexity is particularly notable for tree-based models and ensembles of trees, such as sums of trees. In the context of trees and sums of trees, the computational complexity of exact SHAP value computation is exponential. However, TreeExplainer employs an efficient algorithm that reduces this complexity to a low-order polynomial. This reduction in complexity makes the computation of SHAP values more feasible and efficient for practical use with tree-based models. Furthermore, the method takes advantage of the linearity property of SHAP values. Specifically, when combining or summing the SHAP values of two individual functions (such as two trees or two components of an ensemble), the resulting SHAP values are simply the sum of the SHAP values of the original functions. By reducing the computational complexity, TreeExplainer enables the practical application of SHAP values to larger and more complex tree-based models, making their interpretability and feature importance analysis more feasible in real-world scenarios.

Additionally, it is worth noting that the SHAP framework, including TreeExplainer, does not rely on the assumption of feature independence. This is an important characteristic, as many real-world datasets often exhibit complex interdependencies among features. The SHAP approach provides a flexible

This article has been accepted for publication in IEEE Transactions on Emerging Topics in Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TETC.2024.3439884

TABLE III
DETECTION RESULTS

Dataset	Accuracy	Precision	Recall	F1-Score
fla	0.997	0.994	0.978	0.986
bcf	0.986	0.982	0.904	0.941
sub	0.919	0.880	0.393	0.543
binary mixed	0.925	0.952	0.833	0.889
multi partial	0.916	0.914	0.916	0.903
multi all	0.919	0.918	0.919	0.908

and robust methodology for interpreting model predictions, capturing the effects of feature interactions and dependencies without imposing restrictive assumptions.

V. RESULTS

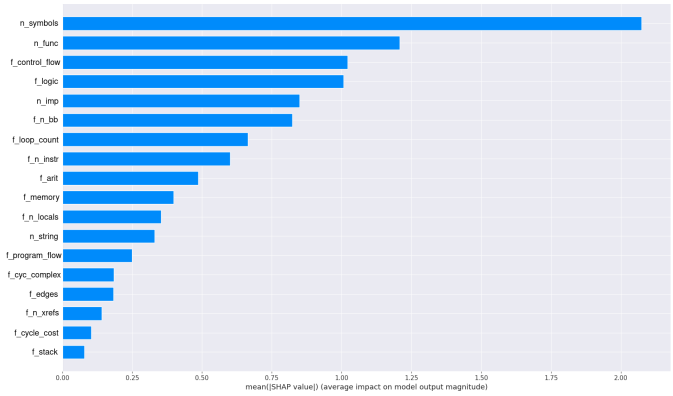

Fig. 2. *Global Feature Importance: Bogus Control Flow Transformation*

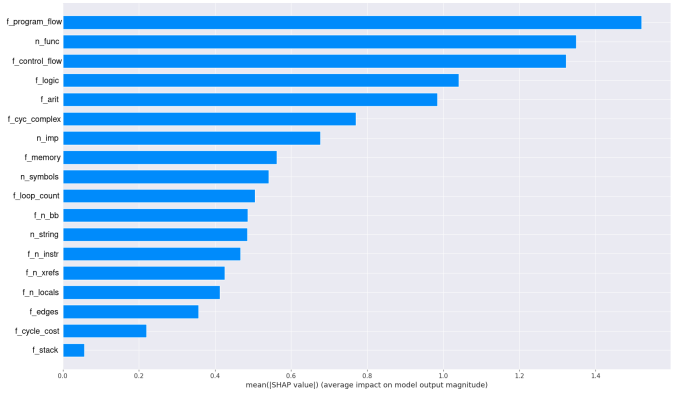

Fig. 3. *Global Feature Importance: Flattening Transformation*

The results of our obfuscation detection process are shown in Table III. In order to evaluate the quality of our analysis we employed several metrics, such as *Accuracy, Precision, Recall*, and *F1-score*. Specifically, such measures are defined as: (1) *Accuracy*, formally defined as $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$, represents the percentage of correctly classified samples. (2) *Precision*, formally defined as $Precision = \frac{TP}{TP+FP}$, represents the number of true positive instances divided by the total number of instances predicted as positive. (3) *Recall*, formally defined as $Recall = \frac{TP}{TP+FN}$, represents the fraction of true positive instances over the total number of actual positive instances. (4) *F1-score*, formally defined as $F1 = \frac{2 \cdot (Precision \cdot Recall)}{Precision+Recall}$, represents the harmonic mean of precision and recall, balancing both metrics. The above formulas are defined in terms of: True Positive (TP), which is the number of the correctly predicted positive instances, True Negative (TN), which is the number of correctly predicted negative instances, False Positive (FP), which is the number of the incorrectly predicted positive instances, and False Negative (FN) is the number of the incorrectly predicted negative instances.

Promising results have been achieved for all datasets,

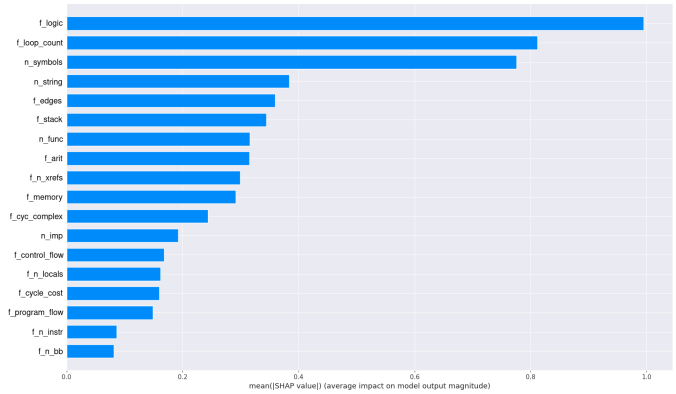

Fig. 4. *Global Feature Importance: Instruction Substitution Transformation*

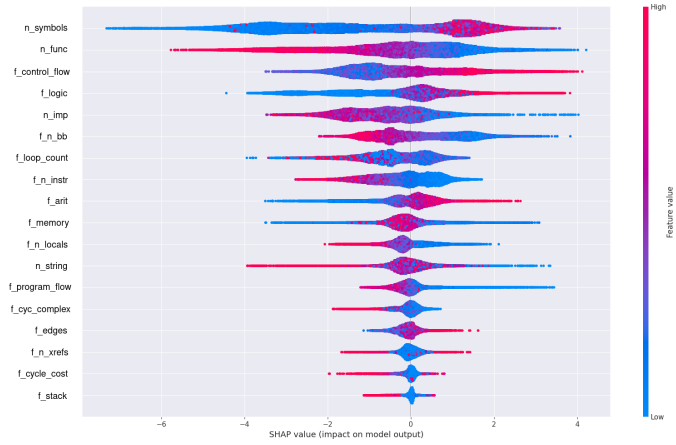

Fig. 5. *Beeswarm Plot: Bogus Control Flow Transformation*

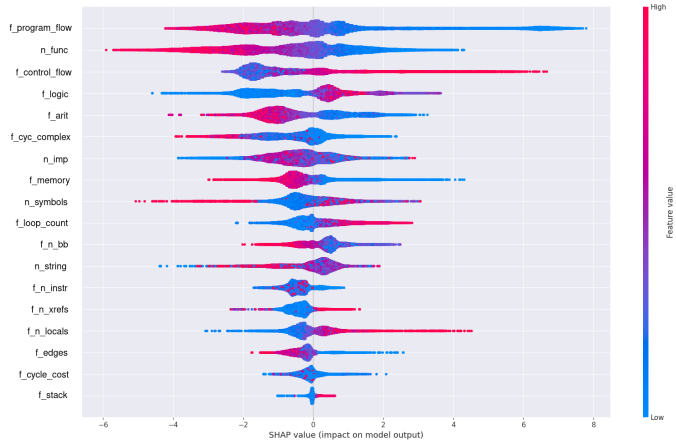

Fig. 6. *Beeswarm Plot: Flattening Transformation*

This article has been accepted for publication in IEEE Transactions on Emerging Topics in Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TETC.2024.3439884

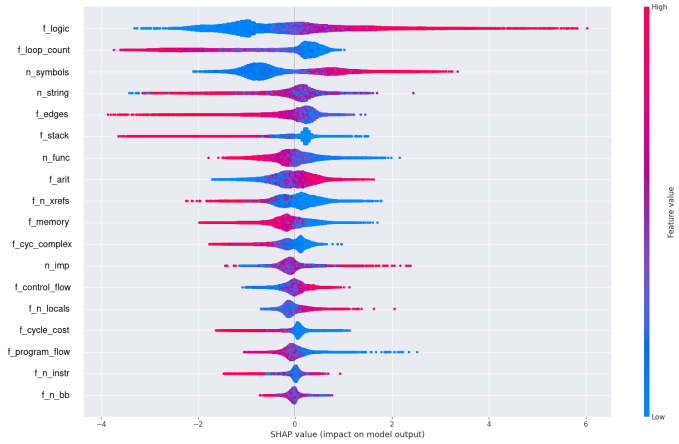

Fig. 7. *Beeswarm Plot: Instruction Substitution Transformation*

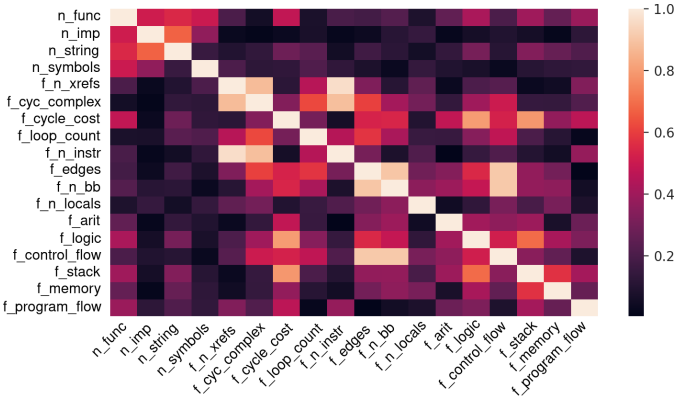

Fig. 8. *Correlation Plot: Bogus Control Flow Transformation*

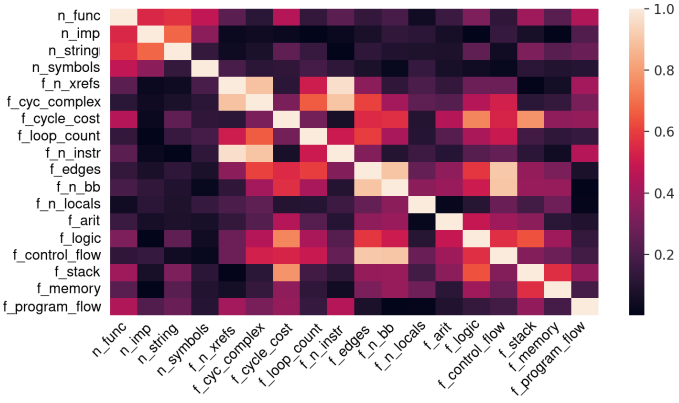

Fig. 9. *Correlation Plot: Flattening Transformation*

thereby proving the effectiveness of performing obfuscation detection relying on a machine learning (ML) based model that leverages static features. The only suboptimal result was observed when handling the `sub` dataset, as indicated by a recall value of 0.393, suggesting that a significant number of positive instances were not correctly identified by the model. However, this outcome was expected. According to

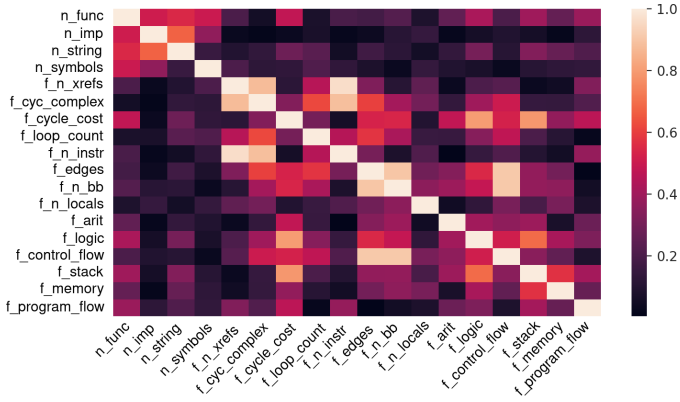

Fig. 10. *Correlation Plot: Instruction Substitution Transformation*

the LLVM Obfuscator webpage*, the instruction substitution transformation primarily operates on arithmetic instructions by making slight modifications. Since the functions analyzed in our experimental evaluation exhibit a scarcity of purely arithmetic operations, obtaining satisfactory results was challenging. Despite the robust evidence provided by the results, demonstrating the worth and success of our obfuscation detection methodology, our study goes beyond simple detection performance. It aims to gain a deeper understanding of the obfuscation process by observing the influence of individual obfuscating transformations on features, and utilizing this information to detect obfuscation or identify invariant features. Specifically, invariant features hold significant value as they may be employed to create malware signatures that are obfuscation-resilient.

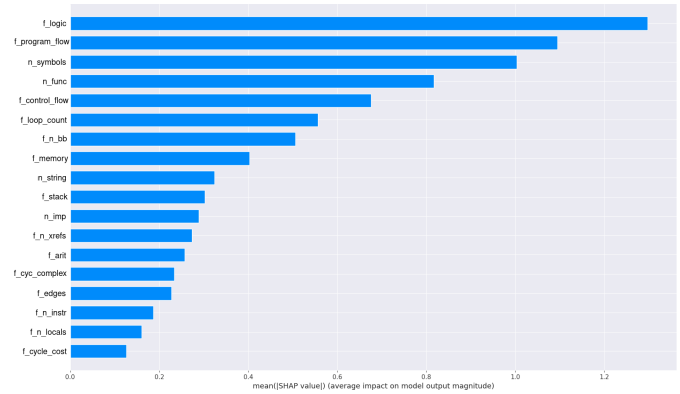

Fig. 11. *Global Feature Importance:* `binary mixed` *dataset*

Initially, we focus on the binary classification task on the datasets associated with a specific transformation. The distinct features involved in each obfuscating transformation, as depicted in Figures 2, 5, 3, 6, 4 and 7 offer significant insights. It is particularly interesting to observe that certain features are significant only when they assume low or high values. For instance, consider the cyclomatic complexity, which measures the number of linearly independent paths through a function's source code. Figure 6 demonstrates that when the cyclomatic

*https://github.com/obfuscator-llvm/obfuscator/wiki/Instructions-Substitution

This article has been accepted for publication in IEEE Transactions on Emerging Topics in Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TETC.2024.3439884

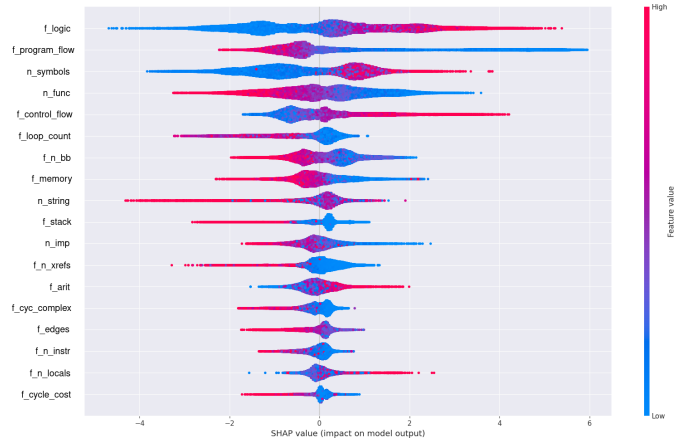

Fig. 12. *Beeswarm Plot:* `binary mixed` *dataset*

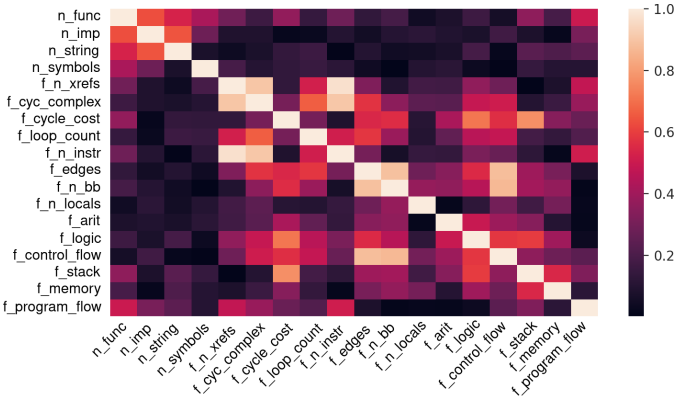

Fig. 13. *Correlation Plot:* `binary mixed` *dataset*

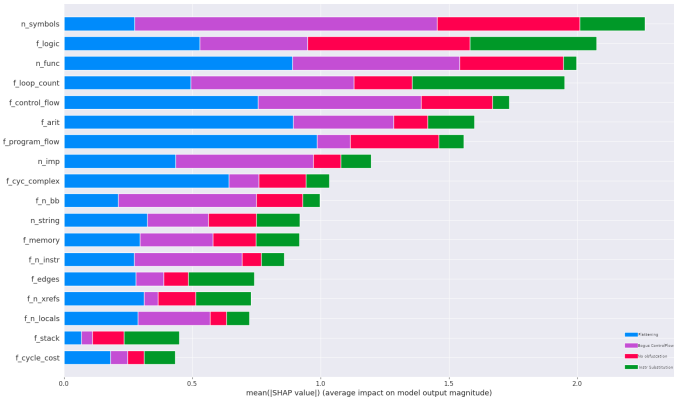

Fig. 14. *Global Feature Importance:* `multi partial` *dataset*

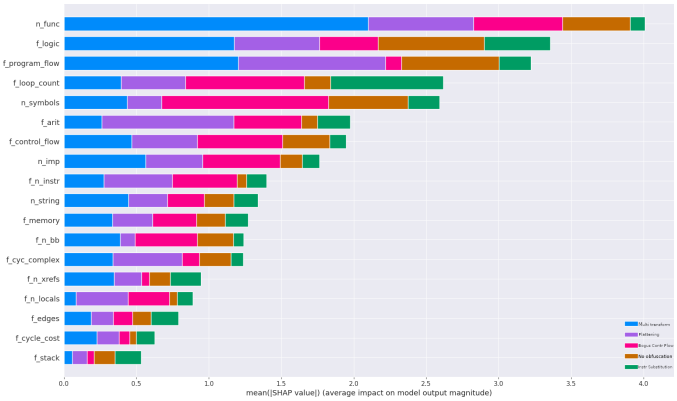

Fig. 15. *Global Feature Importance:* `multi all` *dataset*

complexity value is high, there is a high likelihood that no flattening has been applied to the target program. However, when its value is low, we are not able to make the inverse claim. This outcome obtained through the XAI procedure carries considerable meaning as it allows to determine which obfuscating transformation was implemented by examining the features identified as having the greatest impact for a given sample. This information is extremely precious in both the domains of de-obfuscation and reverse engineering.

The results of the correlation matrices for the first three datasets presented in Figures 8, 9 and 10, generated using the SHAP technique, suggest a low level of correlation among the features in the datasets. Notably, the similarity between the control transformation experiments is reflected in these matrices. The low level of correlation suggests that the features are relatively independent, providing unique information about the investigated transformations.

Shifting our focus to the `binary mixed` dataset, it is crucial to note that despite the presence of multiple obfuscating transformations, the key features that predominantly characterize the obfuscated software of the previous experiments remain consistent, as shown in Figures 11, 12 and 13. We observe that the number of symbols and the feature related to logic and program flow instructions, which serve as primary features in analyzing individual obfuscating transformation, also emerge as main features in the experiment conducted on the `binary mixed` dataset. This observation suggests that even in the presence of multiple transformations, their effects appear to accumulate, without one transformation adversely affecting the outcomes of the others.

The most remarkable results are observed in the multi-classification experiments. In these experiments, we aim not only to determine whether a binary has been obfuscated or not, but also to identify the specific type of obfuscating transformation applied. Table III presents the excellent classification results achieved by our methodology. By examining the features involved in the classification, as shown in Figures 14 and 15, we observe that the characteristic features identified in the binary classification experiments also play a significant role in the multi-classification experiments.

Figure 14 shows the results of the experiments conducted on the `multi partial` dataset, which comprises binaries obfuscated using a single obfuscating transformation per binary. In Figure 15, we observe the results of the experiments performed on the `multi all` dataset, where binaries are obfuscated using three different obfuscating transformations simultaneously. Notably, even in the presence of multiple transformations, the accuracy, precision, recall, and F1-score achieved are remarkably satisfactory. These results confirm our assumption that detecting obfuscating transformations in binaries can be effectively accomplished through a machine learning-based approach, leveraging features extracted via

static binary analysis.

VI. Conclusions

This paper aims to contribute to the field of obfuscation detection by providing insights into this important topic. We employ an ML-based approach, augmented with XAI, to gain a comprehensive understanding of how obfuscating transformations affect the properties of target binaries. The obtained results not only confirm our initial hypotheses but also present numerous intriguing avenues for further exploration by the malware research and reverse engineering communities.

In our future work, we aim to broaden the scope of our analysis by exploring additional obfuscating transformations and features, leading to a more comprehensive assessment of obfuscation techniques.

References

[1] D. Quarta, M. Ianni, A. Machiry, Y. Fratantonio, E. Gustafson, D. Balzarotti, M. Lindorfer, G. Vigna, and C. Kruegel, "Tarnhelm: Isolated, transparent & confidential execution of arbitrary code in arm's trustzone," in *Proceedings of the 2021 Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*, ser. Checkmate '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 43–57. [Online]. Available: https://doi.org/10.1145/3465413.3488571

[2] R. Honick, *Software piracy exposed*. Elsevier, 2005.

[3] C. Greco, M. Ianni, A. Guzzo, and G. Fortino, "Explaining binary obfuscation," in *2023 IEEE International Conference on Cyber Security and Resilience (CSR)*, July 2023, pp. 22–27.

[4] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[5] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 2007, pp. 231–245.

[6] J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004.

[7] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, 2007.

[8] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, "Control flow graphs as malware signatures," in *International workshop on the Theory of Computer Viruses*, 2007.

[9] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. IEEE, 2010.

[10] M. Bat-Erdene, T. Kim, H. Li, and H. Lee, "Dynamic classification of packing algorithms for inspecting executables using entropy analysis," in *2013 8th International Conference on Malicious and Unwanted Software:" The Americas"(MALWARE)*. IEEE, 2013, pp. 19–26.

[11] M. Driller, "Metamorphism in practice," *29A Magazine*, vol. 1, 2002.

[12] D. Mohanty, "Anti-virus evasion techniques and countermeasures," *Published online at http://www. hackingspirits. com/ethhac/papers/whitepapers. asp. Last accessed on*, vol. 18, 2005.

[13] Rajaat, "Polimorphism," *29A Magazine*, vol. 1, no. 3, 1999.

[14] L. Julus, "Metamorphism," *29A Magazine*, vol. 1, no. 5, 2000.

[15] M. Ianni, E. Masciari, and D. Saccà, "An overview of the endless battle between virus writers and detectors: How compilers can be used as an evasion technique," in *Proceedings of the 8th International Conference on Data Science, Technology and Applications, DATA 2019, Prague, Czech Republic, July 26-28, 2019.*, 2019, pp. 203–208. [Online]. Available: https://doi.org/10.5220/0007922802030208

[16] M. Schiffman, "A brief history of malware obfuscation: Part 1 of 2," Published online at https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2, accessed: 2018-11-13.

[17] ——, "A brief history of malware obfuscation: Part 2 of 2," Published online at https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_2_of_2, accessed: 2018-11-13.

[18] W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, 2006.

[19] P. Szor and P. Ferrie, "Hunting for metamorphic," in *Virus bulletin conference*. Prague, 2001.

[20] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES, Tech. Rep., 2006.

[21] E. Konstantinou and S. Wolthusen, "Metamorphic virus: Analysis and detection," *Royal Holloway University of London*, vol. 15, p. 15, 2008.

[22] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," *CS701 Construction of compilers*, vol. 19, 2005.

[23] L. S. Shapley, *A Value for n-Person Games*. Princeton University Press, 1953.

[24] A. Salem and S. Banescu, "Metadata recovery from obfuscated programs using machine learning," in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, ser. SSPREW '16. New York, NY, USA: Association for Computing Machinery, 2016.

[25] R. Tofighi-Shirazi, I. M. Asăvoae, and P. Elbaz-Vincent, "Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning," in *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*, ser. SSPREW9 '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3371307.3371313

[26] S. Jiang, Y. Hong, C. Fu, Y. Qian, and L. Han, "Function-level obfuscation detection method based on graph convolutional networks," *Journal of Information Security and Applications*, vol. 61, 2021.

[27] A. J. Smith, R. F. Mills, A. R. Bryant, G. L. Peterson, and M. R. Grimaila, "Redir: Automated static detection of obfuscated anti-debugging techniques," in *2014 International Conference on Collaboration Technologies and Systems (CTS)*. IEEE, 2014, pp. 173–180.

[28] D. Kim, A. Majlesi-Kupaei, J. Roy, K. Anand, K. ElWazeer, D. Buettner, and R. Barua, "Dynodet: Detecting dynamic obfuscation in malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 97–118.

[29] S. Treadwell and M. Zhou, "A heuristic approach for detection of obfuscated malware," in *2009 IEEE International Conference on Intelligence and Security Informatics*. IEEE, 2009, pp. 291–299.

[30] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed. IEEE, 2015, pp. 3–9.

[31] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *2001 International Conference on Dependable Systems and Networks*. IEEE, 2001, pp. 193–202.

[32] T. László and Á. Kiss, "Obfuscating c++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, no. 1, pp. 3–19, 2009.

[33] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in neural information processing systems*, 2017.

[34] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.

[35] ——, "Anchors: High-precision model-agnostic explanations," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[36] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[37] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S.-I. Lee, "From local explanations to global understanding with explainable ai for trees," *Nature Machine Intelligence*, vol. 2, no. 1, pp. 2522–5839, 2020.

VII. Biography Section

This article has been accepted for publication in IEEE Transactions on Emerging Topics in Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TETC.2024.3439884

Claudia Greco is a Ph.D. student at the Department of Computer Science, Modeling, Electronic and System Engineering (DIMES) of the University of Calabria, Italy. She is actively engaged in research in the field of cybersecurity. Her primary research interests lie in the areas of Binary Analysis, Obfuscation and IoT Security. She received a Master's degree from the University of Calabria after completing a research period at Northeastern University in Boston, MA (USA) for her thesis. During her Ph.D. studies, she was a visiting researcher at Wien Universität in Vienna (Austria).

Michele Ianni is an Assistant Professor at the Department of Computer Science, Modeling, Electronic and System Engineering (DIMES) of the University of Calabria, Italy. He received a Ph.D. degree in Information and Communication Technologies from the University of Calabria, Italy, in 2018. During his Ph.D. he was a visiting researcher in SecLab, University of California, Santa Barbara. Previously, he was a Postdoctoral Researcher at the University of Calabria, Italy and at the University of Verona, Italy. In 2023, he was a visiting professor at the Instituto Superior Técnico of the University of Lisbon, Portugal. Michele Ianni's main research interests include Binary Analysis and Exploitation, Obfuscation, Watermarking, Malware, Trusted Execution Environments and IoT Security.

Antonella Guzzo (IEEE member) is an Associate Professor at the DIMES Department, University of Calabria, Italy. Previously, she was a research fellow in the High Performance Computing and Networks Institute (ICAR-CNR), National Research Council, Italy. Her research interests include Process Mining, Data Mining, Artificial Intelligence and Deep Learning. She authored more than 75 papers in top international journals and conferences. She serves as reviewer for over 20 international journals and as Program Committee (PC) member in many international conferences and workshops. Since 2005 she has been involved in national and international research projects as member and/or responsible of unit and/or responsible of project. She is member and cofounder of the IEEE task force in process mining and she is member of the IEEE Working Group for the definition of the Standard XES - eXtensible Event Stream.

Giancarlo Fortino (FIEEE) received a Ph.D. degree in systems and computer engineering from the University of Calabria (Unical) Italy, in 2000. He is a Full Professor of Computer Engineering with the Department of Informatics, Modeling, Electronics, and Systems, Unical. He is a Highly Cited Researcher 2002–2023 by Clarivate in Computer Science. His research interests include wearable computing systems, IoT, and cybersecurity. He published more than 650 articles in premiere journals, conferences, and book series. He is a (Founding) Series Editor of the IEEE Press Series on Human-Machine Systems and the Internet of Things series (Springer) and an associate editor (AE) of premier IEEE Transactions.

He is a cofounder and the CEO of SenSysCal S.r.l., Rende, Unical spinoff focused on innovative Internet-of-Things (IoT) systems, and a cofounder of BigTech S.r.l., Cosenza, Italy, a startup focused on artificial intelligence (AI)-driven systems and big data.