الجمهورية الجزائرية الديمقراطية الشعبية
Peoples Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University May 8, 1945 -Guelma -

Faculty of Mathematics, Computer Science and Material Sciences
Computer Science Department

Masters Degree Thesis
Branch: Computer Science
Option: Information and Communication Sciences and Technologies

Theme:

A code refactoring and machine learning based approach for the detection of PHP malware

Presented by:
Oussama Mrabet

Jury Members:

Title	Full Name	Quality
Dr.	Lazhar Farek	Chairperson
Dr.	Abdelhakim Hannousse	Supervisor
Dr.	Chemseddine Chohra	Examiner

September 2024

Acknowledgements

— First and foremost, I would like to express my deepest gratitude to Allah for granting me the strength, knowledge, and perseverance to complete this thesis. Without His blessings, this achievement would not have been possible.

I am profoundly grateful to the ones who helped through the path, to family and friends who helped.

I extend my sincere thanks to the department staff, especially to the head of the department, Professor Zineddine Kouahla. Words cannot adequately express my appreciation for his unwavering support and dedication to the students over the years. His leadership has been a cornerstone of our academic journey.

I would also like to acknowledge the other outstanding professors and PhD students who have always been there for us: Professor Chohra, Professor Farou, Professor Seridi, Professor Halimi, Professor Benhamza, Professor Abdelmoumen, and particularly Miss Aggoune. Your guidance and support have been invaluable.

Lastly, I want to express my heartfelt thanks to my colleagues. To the younger ones, whom we are going to miss dearly, and to the older ones, whose absence we already feel, thank you for the camaraderie and shared experiences that made this journey memorable. The Refactoring code and updated refactoring algorithms were made by help of Allah. Sincerely, M'rabet Oussama

Abstract

The proliferation of malware has necessitated advanced techniques for detection and prevention. This research investigates the impact of refactoring PHP code on the effectiveness of machine learning algorithms in detecting malware. By manipulating the PHP Abstract Syntax Tree (AST), various refactoring patterns were applied to generate a cleaner code base. The study compares the performance of multiple machine learning models, including Bernoulli Naive Bayes, KNeighbors, Decision Tree, Logistic Regression, SVM, Random Forest, AdaCost, and Gradient Boosting Decision Trees, before and after code refactoring. Results indicate significant improvements in recall, precision, accuracy, and F1 score post-refactoring, demonstrating the potential of AST-based code manipulation in enhancing malware detection.

Keywords: PHP malware detection, Code refactoring, Machine learning.

Table of Contents

List of Figures

List of Tables

Introduction

Malware detection in PHP code is a critical aspect of cybersecurity, given the widespread use of PHP in web development. As one of the most popular scripting languages for server-side applications, PHP powers millions of websites and web applications globally. This ubiquity, however, makes PHP an attractive target for attackers, who often exploit its vulnerabilities to inject malicious code. In recent years, the sophistication of PHP malware has grown, with many attacks relying on obfuscation and dynamic code execution techniques that evade traditional detection methods. This rise in complexity presents an ongoing challenge for cybersecurity professionals and emphasizes the need for more advanced detection strategies.

A significant issue in PHP malware detection is the difficulty of identifying malicious code within obfuscated or dynamically generated code, which allows malware to hide its true intent from standard analysis tools. Traditional approaches, such as signature-based detection or simple heuristics, often fail to keep up with the rapidly evolving tactics used by malware authors. The sheer variety of PHP malware types, ranging from web shells to code injection attacks, makes the problem even more complex. This research seeks to address these challenges by applying a structural approach to PHP code analysis, focusing on refactoring code to expose hidden malware patterns.

The primary contribution of this thesis is an innovative approach to PHP malware detection using the Abstract Syntax Tree (AST) to refactor and simplify PHP code. By transforming PHP code into its AST representation, it becomes possible to restructure the code in a way that reveals malicious behaviors that might otherwise be concealed. This study proposes that by preprocessing code through refactoring, machine learning algorithms can more effectively detect malware, even in cases where the original code was obfuscated. Through the generation of a cleaned and refactored dataset, this research aims to assess the impact of these techniques on the performance of various machine learning classifiers.

The remainder of this thesis is organized as follows: Chapter 1 provides a comprehensive overview of PHP malware, discussing how PHP works, the types of malware commonly encountered, and the challenges in detecting these threats. Chapter 2 presents the proposed system architecture for PHP code refactoring , including AST generation and feature extraction techniques. Chapter 3 details the implementation of the system, the datasets

used, and the results of applying different machine learning classifiers to detect malware. Finally, the thesis concludes with a discussion of the findings, their implications for future research, and potential improvements to the proposed approach.

Chapter 1

Understanding and Detecting PHP Malware: A Structural Approach

PHP is a widely-used scripting language for web development, creating dynamic web applications. Its server-side execution model enhances web functionality but also attracts malicious intent, leading to PHP malware. PHP malware includes various malicious scripts and payloads that exploit vulnerabilities in PHP-based web applications, such as code injection, backdoors, and web shells. Understanding PHP malware's characteristics and mechanisms is crucial for effective detection and mitigation. This chapter provides an overview of PHP, its role in web development, and its technical aspects, including compilation, abstract syntax tree, and opcode execution. It also explores PHP malware types, motivations, and techniques, setting the stage for discussions on detection methodologies and cybersecurity defenses.

1.1 Introduction to PHP Malware

1.1.1 Overview of PHP

PHP (recursive acronym for PHP: Hypertext Preprocessor) is an open source general-purpose scripting language that is especially used for web development and can be embedded into HTML [4].

PHP differs from client-side JavaScript, for example, in that its code runs on the server and generates HTML that is sent to the client. After the script was performed, the client would get the results, but they wouldn't know what the underlying code was. Even if you set up your web server to use PHP to handle all of your HTML files, users will never be able to discover what tricks you have up your sleeve.

The nicest thing about PHP is that, although it offers many advanced capabilities for a professional programmer, it is incredibly simple to use even for beginners. Read through

the extensive list of PHP's features without fear. You may quickly get started and begin writing basic scripts in a few hours.

Figure 1.1 – Usage of php for websites, 10 sep 2024 [54].

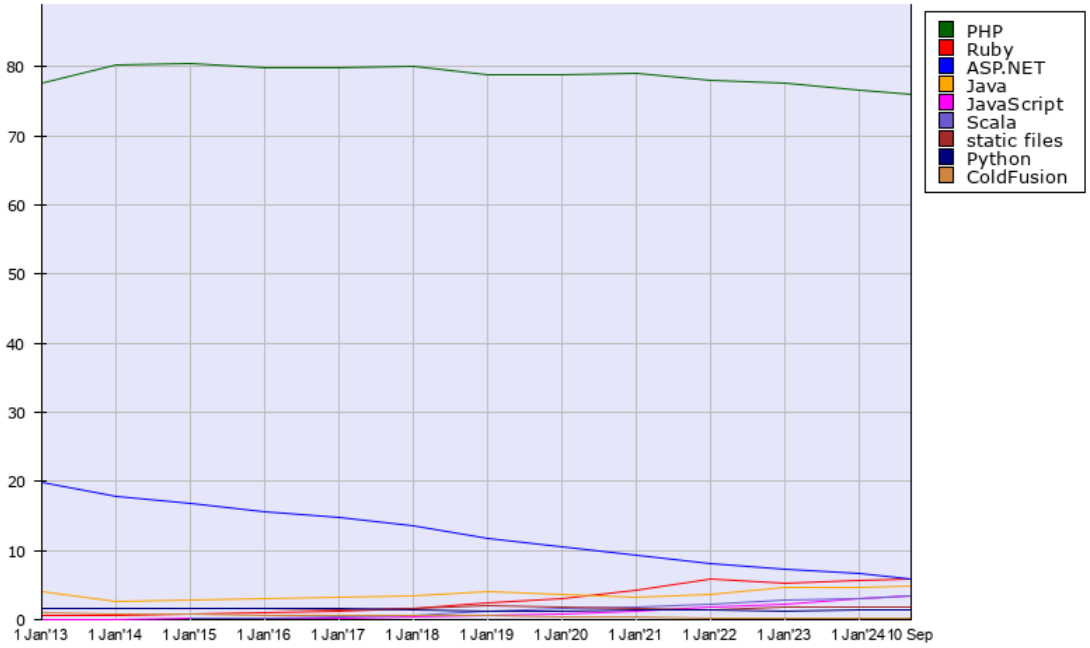

Figure 1.2 – Historical trends in the usage statistics of server-side programming languages for websites chart [52].

1.1.2 HOW DOES PHP WORK (TECHNICAL SIDE AND COMPILING STEPS LEXER,AST,OPCODE)

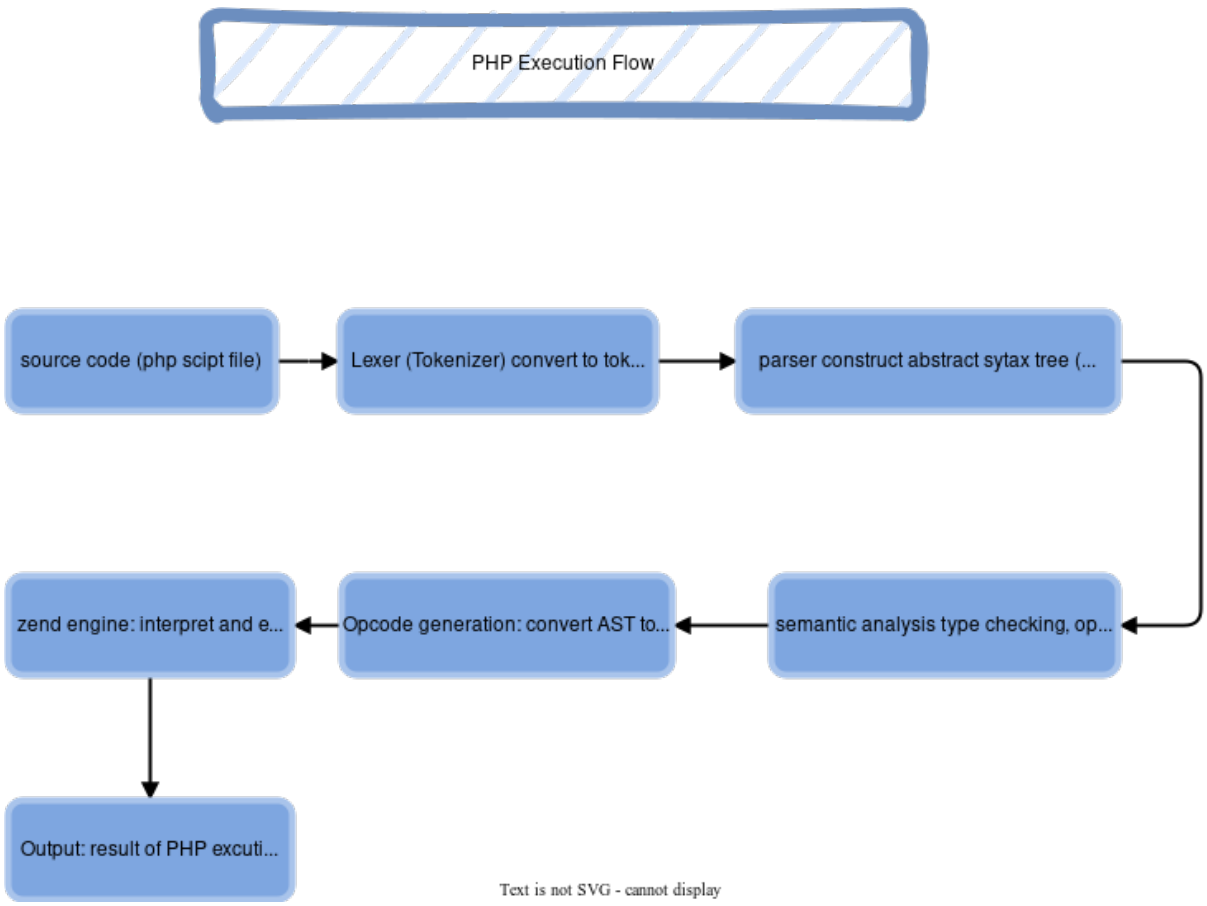

Figure 1.3 – php execution flow

The execution process of PHP involves several key steps that transform high-level source code into the final output displayed or processed by the system. Each step plays a vital role in ensuring the code is executed efficiently and correctly. Below is a detailed breakdown of this workflow.

1. **Source Code**:
   The process begins with the developer writing PHP source code, which consists

of human-readable instructions. This source code defines the logic, functions, and operations that need to be performed by the server.

2. **Lexical Analysis (Lexer)**:
   Once the PHP source code is submitted for execution, the first task is to break it down into smaller components through *lexical analysis*. This process, handled by the *lexer*, converts the code into a series of tokens. These tokens represent distinct elements of the code, such as variables, functions, operators, and syntax symbols.
3. **Abstract Syntax Tree (AST)**:
   After tokenization, the next step is generating an *Abstract Syntax Tree (AST)*. The AST organizes the tokens into a hierarchical structure that reflects the logical flow of the program. By constructing this tree, PHP creates an organized model of the code, which helps in understanding the relationships between different parts of the program.
4. **Semantic Analysis**:
   Once the AST is created, the code undergoes *semantic analysis*. This step ensures that the program makes logical sense. For instance, it checks whether variables are correctly defined before they are used or if functions are called with the appropriate arguments. This analysis helps detect potential errors in the code before it proceeds to the next phase.
5. **Opcode Generation**:
   Following the semantic analysis, the AST is converted into *opcode* (operation code). Opcode is a low-level set of instructions that represent the program in a form that can be understood by the PHP engine. These instructions are much closer to machine code and describe precisely how the program should be executed.
6. **Zend Engine Execution**:
   The generated opcodes are passed to the *Zend Engine*, which is the core component responsible for running PHP scripts. The Zend Engine processes the opcode, executing the instructions in sequence. During this phase, any computations, database interactions, or other logic defined in the source code are carried out.
7. **Output Generation**:
   Finally, once the Zend Engine has processed the opcodes, the resulting *output* is generated. This output could be in various forms, such as HTML code displayed to users in a web browser, or other forms of data returned by the server. The output represents the final product of the PHP scripts execution, based on the original source code.

This multi-step process is fundamental to how PHP interprets and executes scripts. Each phase ensures that the code is validated, optimized, and transformed in a way that allows the system to efficiently generate the desired output. By breaking down the source

code and processing it in this manner, PHP ensures both accuracy and performance in delivering web-based applications.

1.1.3 PHP Malware Types

PHP malware refers to a variety of harmful scripts and payloads that are inserted into PHP code with the intention of taking advantage of security holes, stealing data, or accessing servers without authorization.

Webshells: Give you command and remote access to the compromised server [32].

Backdoors: Give attackers permanent access by evading authentication [45].

Code Injection: Injects malicious code into PHP scripts that already exist [45].

1.1.4 The Effects and Difficulties of PHP Malware Identification:

It can be difficult to identify PHP malware because of:

* Obfuscation: In order to avoid discovery, attackers frequently obfuscate code [38].

* Polymorphism: Malware is able to alter its look without compromising its essential capabilities [29].

* Complexity of PHP: Static analysis is challenging due to PHP's dynamic characteristics [29].

1.2 Methods for Detecting Malware

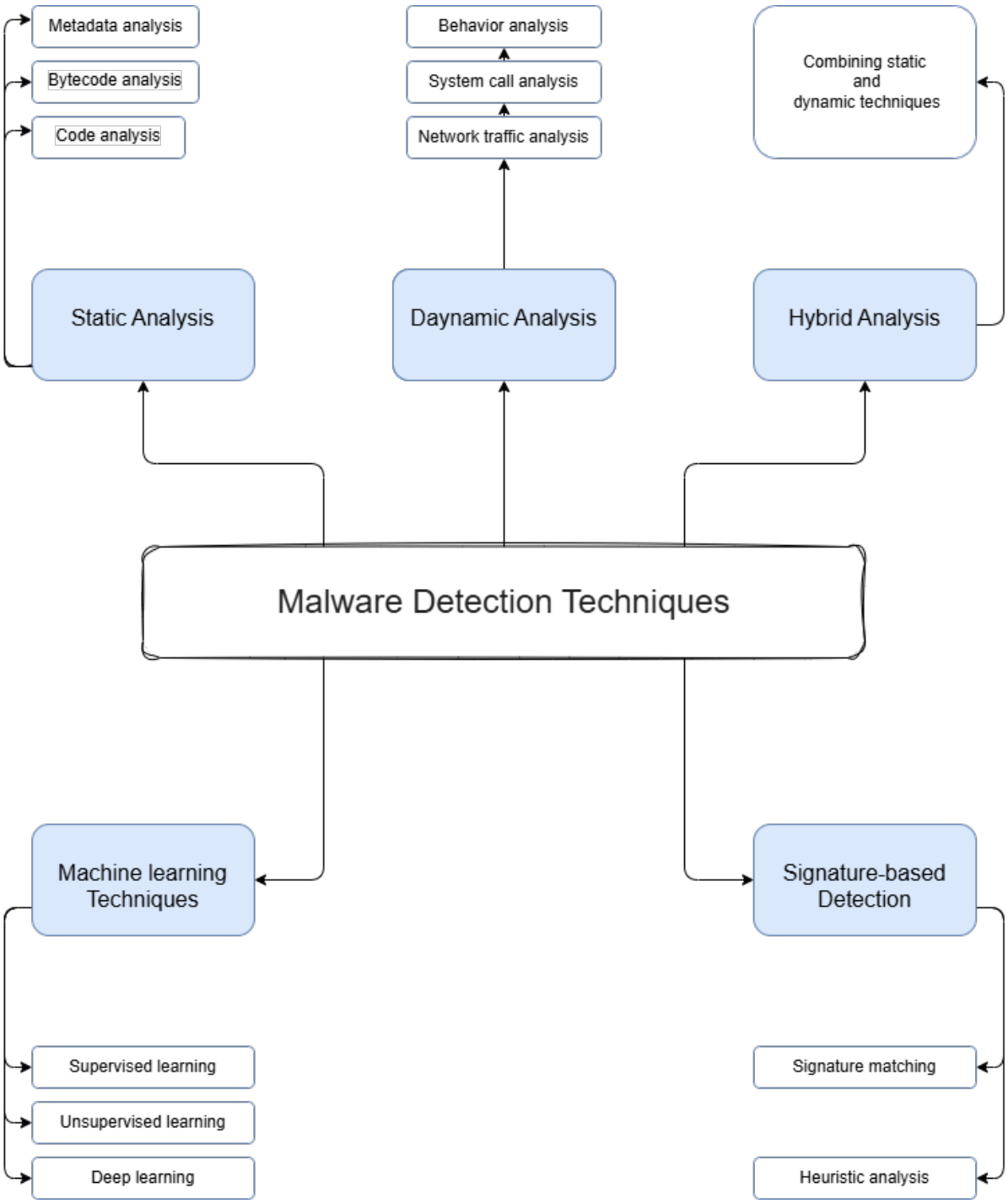

Figure 1.4 – Techniques of Malware Detection

1.2.1 Detection using Signatures

Signature-based detection relies on identifying specific patterns or unique "signatures" within malware code that have already been recognized and cataloged. These signatures

are typically derived from known pieces of malicious code, such as unique strings, sequences of instructions, or byte patterns that are consistently present in certain types of malware.

The process works by scanning files or code for these recognized patterns, and when a match is found, the malware is flagged as a threat. Signature-based detection is highly effective at quickly identifying and neutralizing known threats because it doesn't need to analyze the entire behavior of the malwarejust the signature.

However, this method has a significant limitation: it can only detect malware that has already been analyzed and cataloged in a database. New, unknown, or modified malware can bypass signature detection by altering its code or using techniques like obfuscation, polymorphism, or encryption to disguise its true nature. As a result, while this method is efficient for known threats, it struggles against emerging or evolving malware strains that don't yet have identifiable signatures [7].

1.2.2 Analysis Using Heuristics

Heuristic analysis evaluates code or behavior by examining characteristics and patterns that deviate from normal or expected behavior. Instead of relying on known malware signatures, this method uses predefined rules and algorithms to flag suspicious code or activities based on how malware typically behaves.

For instance, heuristic analysis might look for actions that are unusual or risky, such as attempts to modify critical system files, make unauthorized network connections, or use functions often associated with malware (e.g., self-replication or unauthorized access to sensitive data). By setting these behavioral rules, heuristic systems can identify potentially harmful activities even if the malware is new or unknown.

This approach is particularly effective for detecting emerging threats and variants of existing malware that modify their appearance to evade signature-based detection. Heuristic analysis offers a proactive defense by focusing on how the code behaves rather than what it looks like. However, it can sometimes generate false positives by mistakenly flagging legitimate actions as malicious, requiring a balance between sensitivity and accuracy [23, 25].

1.2.3 Analysis of Behavior

Behavioral analysis involves monitoring programs in real-time as they execute, observing their actions and interactions within the system to detect malicious behavior. Unlike static methods, which analyze the code without running it, behavioral analysis focuses on what the program does while it's running. This approach can uncover hidden malicious activities that may not be visible by simply inspecting the code.

For example, behavioral analysis might look for signs of malware by tracking activities like attempts to modify system files, access unauthorized data, escalate privileges, or

communicate with remote servers. These behaviors are often telltale signs of malware, even if the malware itself uses sophisticated techniques to hide its code or evade detection [36, 56].

1.2.4 Detection Based on Machine Learning

Machine learning-based detection uses advanced algorithms to automatically analyze data and identify patterns or anomalies that indicate the presence of malware. Unlike traditional detection methods, which rely on predefined rules or known signatures, machine learning models learn from large datasets to recognize subtle indicators of malicious activity that may not be immediately obvious.

These algorithms are trained on both legitimate and malicious code, allowing them to distinguish between normal and harmful behavior. As they analyze more data, the models become increasingly proficient at identifying new forms of malware by recognizing similarities to previously encountered threats or unusual patterns in the behavior of programs or network traffic.

One of the key strengths of machine learning-based detection is its ability to adapt and improve over time. As the model is exposed to new data, it can refine its detection capabilities, making it more effective at catching novel or evolving threats, such as zero-day attacks or malware variants that have not been seen before [33, 48].

1.3 web shells

1.3.1 web shells definition:

A web shell can be written in any of the programming languages that the target server supports. These are usually common languages, such as PHP.

1.3.2 Common Techniques used in Web Shells:

Web shells are a type of malicious script that provide attackers with remote access to compromised servers. They are typically written in PHP and allow attackers to execute arbitrary commands, manage files, and perform various harmful operations on the server. To evade detection and maximize their control, attackers employ several common techniques when creating and deploying web shells.

— **Code Obfuscation:** Obfuscation is a method used by attackers to make web shell code harder to read and understand. This can involve renaming variables to meaningless values, using complex code structures, or encoding malicious code. PHP functions like `base64_encode()` and `base64_decode()` are often used to hide the

true intent of the script. This makes it more difficult for automated scanners or human reviewers to detect malicious content at a glance.

— **Encryption and Encoding:** Attackers frequently encrypt or encode parts of the web shell to prevent detection by signature-based tools. For example, the PHP `eval()` function can be used to execute encoded strings dynamically. While this adds a layer of stealth, behavioral analysis tools that monitor for suspicious function calls can still flag such activity.

— **Use of Legitimate File Names:** A popular method to evade detection is to disguise web shells under seemingly legitimate names, such as `index.php` or `config.php`. These names are often chosen to blend in with the typical structure of a web application, making it harder for administrators to recognize the file as malicious.

— **Minimalist Approach:** Some web shells are deliberately kept small and simple to avoid detection. Instead of including many features, they rely on basic command execution, often using just a few lines of code. These minimalist shells are harder to detect due to their limited footprint, but they still provide enough functionality for attackers to upload additional tools or escalate their control.

— **Embedding in Legitimate Code:** Web shells are often embedded within legitimate files to avoid drawing attention. By inserting malicious code into otherwise normal files, attackers make detection more difficult, especially when security systems focus on scanning newly created or modified files.

1.3.3 HISTORY OF WEBSHELLS

One of the earliest known webshells, WSO (Web Shell by oRb), appeared on a Russian hacking forum in 2010. WSO has evolved over the years, adding features and improving its functionality. A newer version, WSO-NG (Next Generation), has enhanced capabilities, including techniques to evade detection and maintain stealth, such as hiding the login page behind a 404 error message and using CSS to move it off-screen. This evolution reflects the increasing sophistication of attackers and their methods [1].

Webshells like WSO-NG are part of a broader ecosystem of tools used by attackers, which include other popular webshells such as China Chopper, B374K, and R57. These tools provide a range of functionalities, from simple file management to advanced reconnaissance and lateral movement within networks [1, 5].

Understanding the history and development of webshells is crucial for developing effective defenses. Organizations must stay vigilant and employ comprehensive security measures to detect and mitigate the threat posed by these powerful tools

1.3.4 WEBSHELL CHARACTERISTICS:

Webshells allow attackers to execute arbitrary commands on the compromised server. This includes running shell commands, interacting with the file system, and manipulating processes. This capability is platform-independent, functioning on both Windows and Linux systems.

Webshells can upload, download, and delete files on the server. They can also create and download archives, which facilitates data exfiltration and the installation of additional malicious software.

Many webshells include functionality to interact with databases, allowing attackers to execute SQL queries, dump database tables, and manipulate database contents. This can lead to data theft or further exploitation of the web application.

Advanced webshells can establish reverse shells, creating a network connection back to the attackers machine. This provides a persistent backdoor for continuous access and control over the compromised server.

Stealth and Evasion:

Webshell traffic often mimics legitimate web traffic, making it difficult to detect. Webshells typically operate over HTTP/HTTPS, blending in with normal web requests. Additionally, attackers may obfuscate webshell code to evade detection by security tools

Customization and Flexibility:

Webshells are highly customizable. Attackers can modify existing webshell scripts or create new ones tailored to specific targets. This flexibility allows webshells to be adapted for various objectives, such as data exfiltration, lateral movement, or persistence

Webshells often include features that help maintain long-term access to the compromised server. This can involve setting up cron jobs, modifying startup scripts, or deploying additional backdoors to ensure the attacker can regain access even after the webshell is detected and removed

Minimal Skill Requirement:

Many webshells are available as ready-to-use scripts on the internet, requiring minimal technical skills to deploy. This accessibility makes them a popular choice among attackers of varying expertise levels

1.3.5 ATTACKERS ADVANTAGES:

Webshell attacks using php are a significant threat in cybersecurity because of their stealth and effectiveness. Here are some advantages and disadvantages of these attacks:

— **Persistence and Control:** Webshells provide attackers with persistent access to a compromised server, allowing them to execute commands, upload or download files, and perform various malicious activities.

— **Stealth:** Webshell traffic often appears as legitimate web traffic, making it difficult to detect. Many webshells operate over HTTPS, encrypting their communications and further concealing their actions from monitoring tools.

— **Flexibility:** Webshells can be customized to suit the attackers needs, from simple command execution to more complex tasks like database manipulation and network connections back to attacker-controlled machines (reverse shells)

— **Ease of Deployment:** Attackers can leverage existing vulnerabilities, such as Local File Inclusion (LFl) or unpatched plugins, to deploy webshells. They can also use publicly available tools to create effective webshells with limited technical skills [6].

1.3.6 STATS ABOUT CURRENT STATE OF WEBSHELLS:

Since January 2023, Talos IR observed an increase in web shell usage from 6% of all threats last quarter to now comprising nearly 25% of all threats. Web shells are malicious scripts that enable threat actors to compromise web-based servers exposed to the internet.(7)This quarter, Talos observed threat actors using publicly available or modified web shells coded in various languages, including PHP, ASP.NET and Perl. After leveraging web shells to establish a foothold and gain persistent access to a system, adversaries remotely executed arbitrary code or commands, moved laterally within the network, or delivered additional malicious payloads. In many of these web shell incidents, adversaries relied heavily on web shell code sourced from publicly available GitHub repositories. This finding is also in line with a trend observed from Talos IR engagements from July to September 2022 (Q3 2022), where adversaries used a variety of open-source tools and scripts hosted on GitHub repositories to support operations across multiple stages of the attack lifecycle [3].

famous webshell attacks incidents:

Recently, an organization in the public sector discovered that one of their internet-facing servers was misconfigured and allowed attackers to upload a web shell, which let the adversaries gain a foothold for further compromise. The organization enlisted the services of Microsoft's Detection and Response Team (DART) to conduct a full incident response and remediate the threat before it could cause further damage [13].

DARTs investigation showed that the attackers uploaded a web shell in multiple folders on the web server, leading to the subsequent compromise of service accounts and domain admin accounts. This allowed the attackers to perform reconnaissance using net.exe, scan for additional target systems using nbtstat.exe, and eventually move laterally using PsExec.

The attackers installed additional web shells on other systems, as well as a DLL backdoor on an Outlook Web Access (OWA) server. To persist on the server, the backdoor implant registered itself as a service or as an Exchange transport agent, which allowed it to access and intercept all incoming and outgoing emails, exposing

sensitive information. The backdoor also performed additional discovery activities as well as downloaded other malware payloads. In addition, the attackers sent special emails that the DLL backdoor interpreted as commands.

The case is one of increasingly more common incidents of web shell attacks affecting multiple organizations in various sectors

This collaborative CSA, which was released by the FBI and the CISA, is intended to spread the word about known CLOP ransomware IOCs and TTPs that have been found through FBI investigations dating back to June 2023.

Open source data indicates that on May 27, 2023, the CLOP Ransomware Gang, also known as TA505, started taking advantage of a SQL injection vulnerability (CVE-2023-34362) in Progress Software's MOVEit Transfer managed file transfer (MFT) software that had not previously been discovered. A web shell known as LEMURLOOT affected MOVEit Transfer web apps that were accessible over the internet, and this allowed for the theft of data from the underlying MOVEit Transfer databases. Similar waves of activity were seen in 2020 and 2021 when TA505 launched campaigns using zero-day exploits against Accellion File Transfer Appliance (FTA) devices and Fortra/Linoma GoAnywhere MFT servers in early 2023 [22].

1.4 ABSTRACT SYNTAX TREE

1.4.1 DEFINITION

An Abstract Syntax Tree (AST) is a tree representation of the syntactic structure of source code written in a programming language. Each node in the tree denotes a construct occurring in the source code. ASTs are used in compilers and interpreters to analyze the syntax of code, optimize performance, and generate executable code. They simplify the complexity of source code by representing nested structures and operations, making it easier for tools to perform tasks like code analysis, transformation, and error checking [2].

1.4.2 APPLICATIONS OF ASTS IN PHP

1. Code Analysis and Linters: o ASTs are used to analyze PHP code for syntax errors, code style issues, and potential bugs. Tools like PHPStan and Psalm use ASTs for static analysis to ensure code quality and adherence to standards.

2. Refactoring Tools:

o ASTs assist in automating code refactoring tasks, such as renaming variables, extracting methods, and restructuring code without changing its behavior. Tools like Rector utilize ASTs for this purpose.

3. Code Generation and Optimization:

o ASTs are employed to transform PHP code into optimized executable code. Compilers and interpreters generate intermediate representations of the code to improve performance.

4. Security Analysis: o ASTs help identify security vulnerabilities by analyzing code paths and data flows, detecting issues like SQL injection or cross-site scripting (XSS).

5. Custom Parsers and Compilers: o Developers can create custom parsers and compilers for domain-specific languages (DSLs) within PHP applications, using ASTs to translate high-level language constructs into executable code.

1.5 STATS ABOUT THE OBFUSCATION OF MALICIOUS CODE

According to a 2020 report by Sucuri, 71% of websites infected with malware were built on PHP.[49] A 2019 study by SANS Institute found that 64% of malware incidents involved PHP-based web applications [30].

Statistic	Value	Year
Percentage of websites infected with malware that use PHP	71%	2020
Percentage of malware incidents involving PHP-based web applications	64%	2019

websites infected with malware that malware incidents involving PHP-based web the statistics of the use of obfuscation with malicious and benign php codes:

— **Malicious PHP Code:** According to a 2019 report by SANS Institute, 64% of malware incidents involved PHP-based web applications [30]. A 2020 analysis by Malwarebytes found that 55% of PHP malware used obfuscation techniques [34].

— **Benign PHP Code:** A 2018 study by PHP-FIG found that 45% of PHP developers use obfuscation techniques for legitimate purposes, such as code protection and intellectual property protection.[42] e According to a 2020 survey by JetBrains, 27% of PHP developers use obfuscation tools to protect their code [31].

1.6 CONCLUSION

PHP malware poses a significant threat to web application security, with diverse types of malware exploiting PHP's vulnerabilities. Detecting PHP malware is challenging due to obfuscation, polymorphism, and dynamic code execution, limiting traditional signature-based approaches. Understanding PHP's technical aspects, such as AST, compilation, and opcode execution, is crucial for developing effective defense strategies. To combat PHP malware, organizations must prioritize security best practices, collaborate with cybersecurity professionals, and leverage advanced detection methodologies. By doing so, stakeholders can safeguard PHP-based applications and ensure a secure web environment.

Chapter 2

PHP Code Refactoring For Malware Detection

While PHP is recognized as the most widely utilized language for web applications, as discussed in Chapter 1, it constitutes over 76.2% of websites where the server-side programming language is known [53]. This extensive use makes PHP-based web applications a significant target and a substantial attack surface for cybercriminals who deploy malicious code. Traditional malware detection methods often encounter challenges due to these criminals' attempts to circumvent detection techniques. One prevalent method employed to bypass traditional detection is obfuscation. Malware obfuscation conceals the code of a program to render it difficult to detect or comprehend, without altering its functionality. Techniques such as compression, encryption, and encoding are frequently employed, often in combination, to evade cybersecurity defenses [21].

This chapter addresses how refactoring, which enhances code structure and readability without altering its functionality, can facilitate the detection of malicious behavior in PHP code. Our approach involves utilizing static analysis to identify suspicious patterns and anomalies without executing the code. By integrating refactoring with feature extraction and machine learning, we aim to improve the accuracy of malware detection by making the original code more comprehensible. This methodology seeks to enhance the security of PHP-based web applications.

2.1 Motivation

Given that many cybercriminals employ obfuscation to circumvent security systems, it is important to recognize that some developers also use obfuscation to protect their code from theft. Research by Oh, Yang, and Lee [39] demonstrates the use of obfuscation methods to safeguard code properties. This introduces a significant challenge, as traditional detection methods, particularly those based on machine learning, face increased difficulty in distinguishing between malicious and benign code due to obfuscation.

For instance, many datasets utilized for training models predominantly consist of obfuscated code. This prevalence of obfuscated code complicates the extraction of genuine code features that are obscured by the obfuscation techniques. Consequently, some models trained on such datasets may not effectively identify malicious code but rather focus on detecting obfuscated code. Obfuscation thus poses a substantial challenge to web security systems, as the same malicious code can manifest in various forms depending on the obfuscation techniques employed by the cybercriminals. This variability necessitates continuous adaptation by security methods and researchers who strive to keep pace with evolving evasion strategies and emerging malware threats.

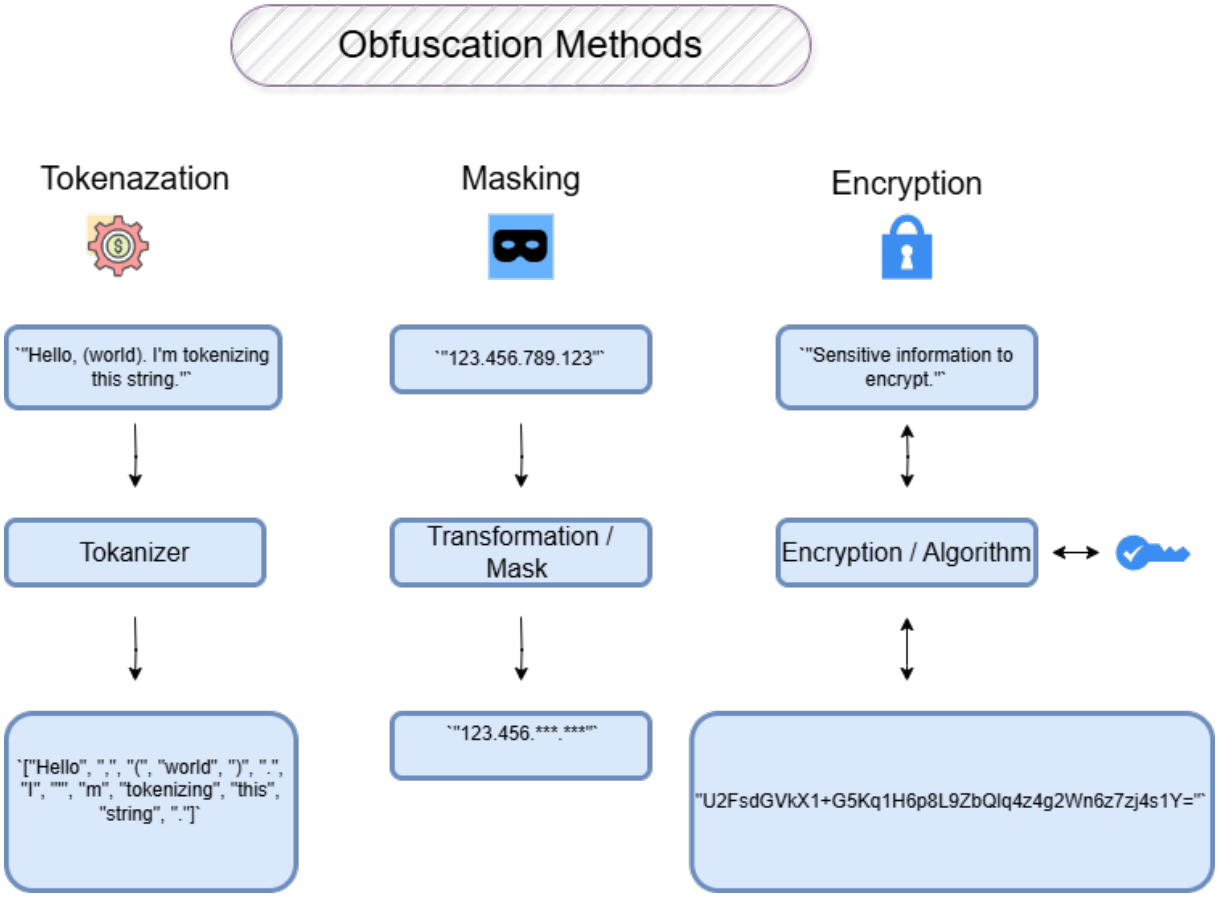

Figure 2.1 – Code Obfuscation methods.

Upon examining the obfuscation methods illustrated in the previous figure **??**, it can be concluded that code refactoring may mitigate the effects of obfuscation, thereby enhancing the readability and clarity of the code for various traditional detection methods. The primary objective of refactoring in this context is to improve code readability, which facilitates the accurate extraction of relevant features while removing misleading elements. Refactoring encompasses more than just deobfuscation; it also involves eliminating unnecessary components, reorganizing elements, and employing other relevant techniques. As

such, refactoring is increasingly recognized as one of the most rapidly advancing areas of research in software engineering [15].

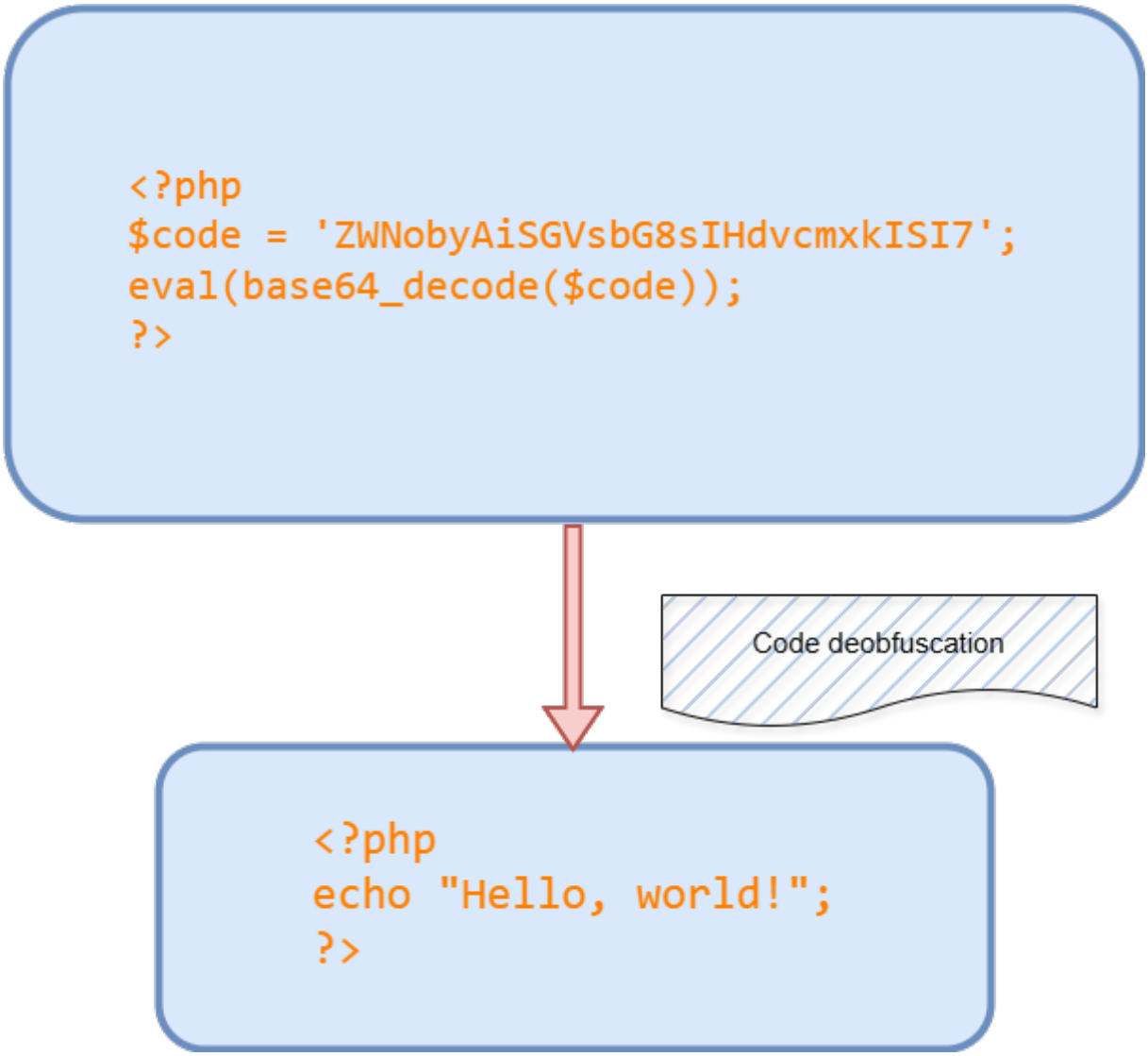

Figure 2.2 – Example of code deobfuscation.

As we observe in Figure2.2, an obfuscated code is hard to understand for human's, and even for machines, even when parsed it doesn't represent the true intentions or functionality of the code like in this case the obfuscation method used is the encryption ("base64_decode") and the when the code gets decrypted it's evaluated through("eval"), which makes it very challenging especially for static methods.

2.2 Proposed System Architecture

In order to address the limitations of the current used methods in detecting PHP malware towards obfuscation and unclean code we are proposing a novel PHP malware detection system, that's main steps are refactoring the code for better feature extraction and training the machine learning model on both refactored and original dataset, then comparing them.

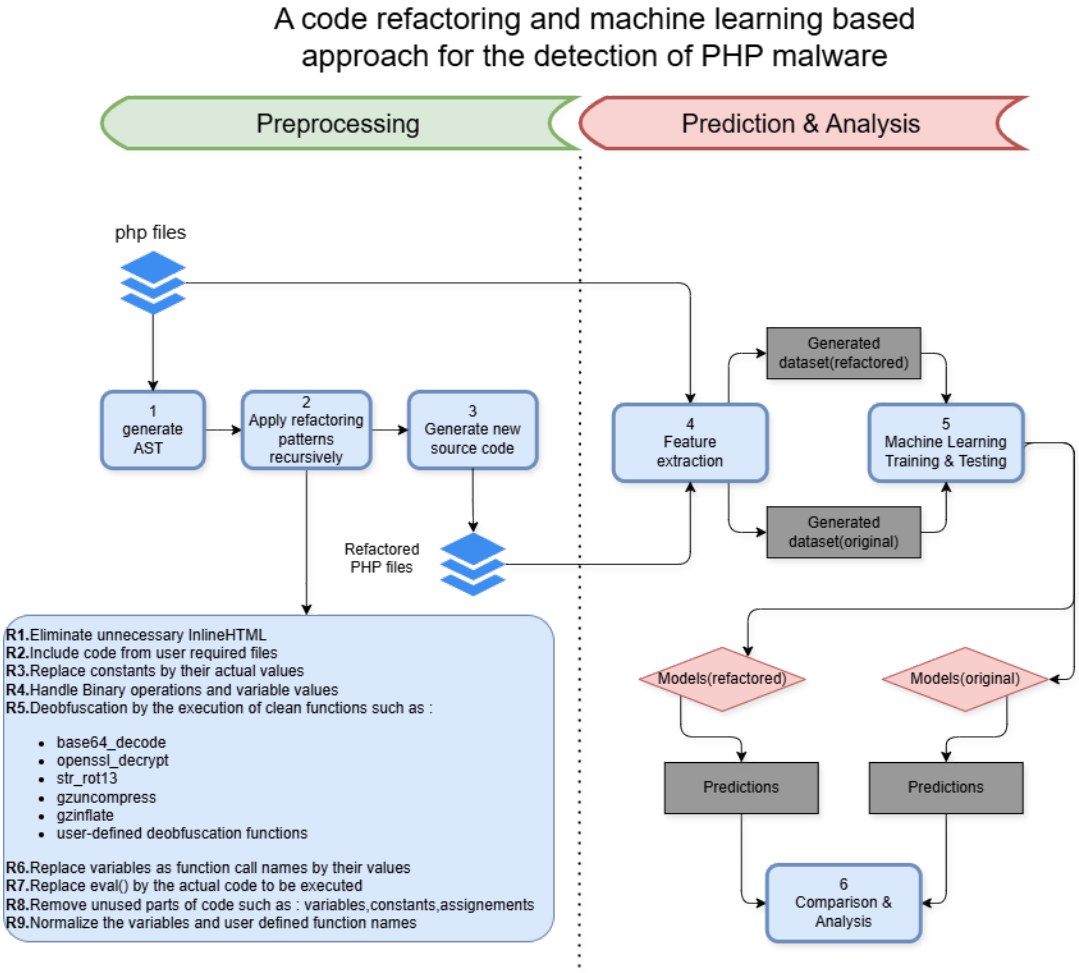

Figure 2.3 – Overall architecture of the proposed PHP malware detection system

As shown in the Figure 2.3 our system starts by gathering the php files dataset and the generating the AST (Abstract Syntax Tree) which best represents the code elements. Parsing the AST alows us to manipulate code element's or we would call them Nodes in the AST. Manipulating the nodes of the AST alows us to apply the rules of refactoring that we chose and ordered.

Following the refactoring of the code and the recursive application of the refactoring rules, new source code is generated from each file's Abstract Syntax Tree (AST). Subsequently, feature extraction is performed on both the refactored and original PHP files. This process yields datasets for each version of the code, which are then utilized to train and

test machine learning algorithms. The models produced from both datasets are compared and analyzed to assess their performance.

The objective of this approach is to enhance the accuracy and optimize the performance of PHP malware detection models. By refining the code and clarifying the tokens, we aim to improve the efficacy of these models.

The proposed PHP malware detection system is organized into several phases, with specific refactoring rules developed based on an analysis of obfuscation methods and PHP AST nodes.

By meticulously following these phases, we establish a structured set of rules and procedures for refactoring. Refactoring involves preserving the functionality of the code while improving its readability and structure. It is crucial to remember that the ultimate goal is to enhance the performance of machine learning models by providing clearer features and making the code's functionality more transparent.

2.2.1 AST Generation

The Abstract Syntax Tree (AST) is a fundamental feature in code representation studies, offering a distinct representation of source code snippets within a specific language and grammatical framework. Due to the highly structured nature of programming languages compared to ordinary text, many studies focused on code seek to reveal the underlying structural information, thereby capturing the syntactic essence of the source code [50].

In our proposed project, we utilize the PhpParser library [44] to parse the ASTs of our PHP files, leveraging its capabilities to analyze and manipulate the structural elements of the code.

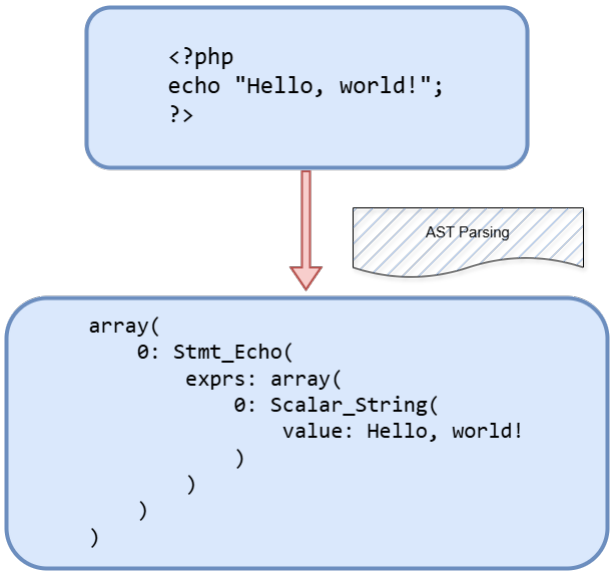

Figure 2.4 – example of parsed AST

As we see the AST consists of a group of nodes,each of them representing an individual and each of them has it's own attributes as show in Figure2.4 depending on the type of node.

Generating AST's helps better manipulate the code, because knowing each node's type and features helps automate the process, while treating the code as plain text to automate the refactoring phase would lead to many challenges and difficulties.

AST has an other advantage which is that it doesn't take into consideration how the code is formated like additional white-spaces and parenthesis [43].

The AST nodes are devided into 2 main types Expressions and Statements.

Expression Nodes

Expr nodes are the elements of code which return or produce a value when evaluated, Such as :

— **ArrayDimFetch**: Represents an array index access.

— **ArrayItem**: Represents an item in an array.

— **Array_**: Represents an array.

— **ArrowFunction**: Represents an arrow function.

— **Assign**: Represents an assignment.

— **BinaryOp**: Represents binary operations like addition, subtraction, etc.

— **BooleanAnd**: Represents a boolean AND operation.

— **BooleanOr**: Represents a boolean OR operation.

— **BooleanNot**: Represents a boolean NOT operation.

— **Cast**: Represents type casting.

— **ClassConstFetch**: Fetches a class constant.

— **Clone_**: Clones an object.

— **ClosureUse**: Represents a variable used in a closure.

— **ConstFetch**: Fetches a constant.

— **Empty_**: Checks if a variable is empty.

— **ErrorSuppress**: Suppresses errors.

— **Eval_**: Represents an eval() statement.

— **FuncCall**: Represents a function call.

— **Include_**: Represents an include or require statement.

— **Instanceof_**: Checks if a variable is an instance of a class.

— **Isset_**: Checks if a variable is set.

— **List_**: Represents a list.

— **MethodCall**: Represents a method call.

— **New_**: Represents a new class instantiation.

— **NullsafeMethodCall**: Calls a method safely on a nullable object.

— **PropertyFetch**: Fetches a property from an object.

— **StaticCall**: Represents a static method call.

— **Ternary**: Represents a ternary conditional operation.

— **Throw_**: Represents a throw statement.

— **Variable**: Represents a variable.

Statement Nodes

Stmt nodes represents elements of code that perform an action but do not necessarily produce a value when evaluated, Such as :

— **Class_**: Represents a class.

— **ClassMethod**: Represents a class method.

— **Const_**: Represents a constant.

— **Declare_**: Represents a declare statement.

— **Echo_**: Represents an echo statement.

— **Function_**: Represents a function.

— **Global_**: Represents a global statement.

— **Goto_**: Represents a goto statement.

— **Interface_**: Represents an interface.

— **Namespace_**: Represents a namespace.

— **Property**: Represents a property.

— **Return_**: Represents a return statement.

— **Trait_**: Represents a trait.

— **TraitUse**: Represents a trait use statement.

— **TryCatch**: Represents a try-catch block.

— **Unset_**: Represents an unset statement.

— **Use_**: Represents a use statement.

Manipulating code as AST vs text

It is essential to acknowledge that parsing and manipulating the Abstract Syntax Tree (AST) can be more demanding in terms of computing resources compared to handling plain text source code. However, the AST provides greater precision in code manipulation and a more structured representation, capturing the semantics of the code more effectively than plain text.

Plain text manipulation may suffice for simple tasks, but it becomes increasingly complex for more involved tasks. For instance, if one aims to replace all occurrences of the base64_decode function call with its returned value, plain text processing might inadvertently modify occurrences of base64_decode within comments or strings, which are not actual function calls. In contrast, AST manipulation allows for the specification of node types, such as distinguishing base64_decode as a FunctionCall, enabling more precise modifications.

Therefore, code refactoring through AST manipulation is more suitable for our proposed system architecture, which involves complex code transformations. Additionally, feature extraction using the AST is more accurate, as it identifies and interacts with the type of each code element with greater precision compared to plain text feature extraction.

Consequently, the initial and crucial step in our proposed system is to generate the AST for each file, thereby preparing it for the application of the proposed refactoring rules.

2.2.2 Code Refactoring

For the generation of the refactored code we have put some rules to optimize the code readability for both humans and machines by studying samples of obfuscated PHP malware codes.Those rules play a big role strengthening its resistance to sophisticated obfuscation methods.

We have put down an algorithm for each rule and ordered them logically to optimize the performance of our system.

— **R1. Eliminate unnecessary InlineHTML**

To enhance the readability and comprehensibility of PHP code while preserving its functionality, refactoring involves maintaining the original code's behavior. A common refactoring practice is to separate PHP code from HTML to clarify the structure of a web application. HTML, being a markup language, is used solely for structuring and presenting content on the web and does not affect the behavior of PHP code.

By separating HTML from PHP code, we delineate the presentation layer (HTML) from the logic layer (PHP). This separation of concerns is a well-established practice in web development, aimed at creating cleaner, more maintainable, and easier-to-debug

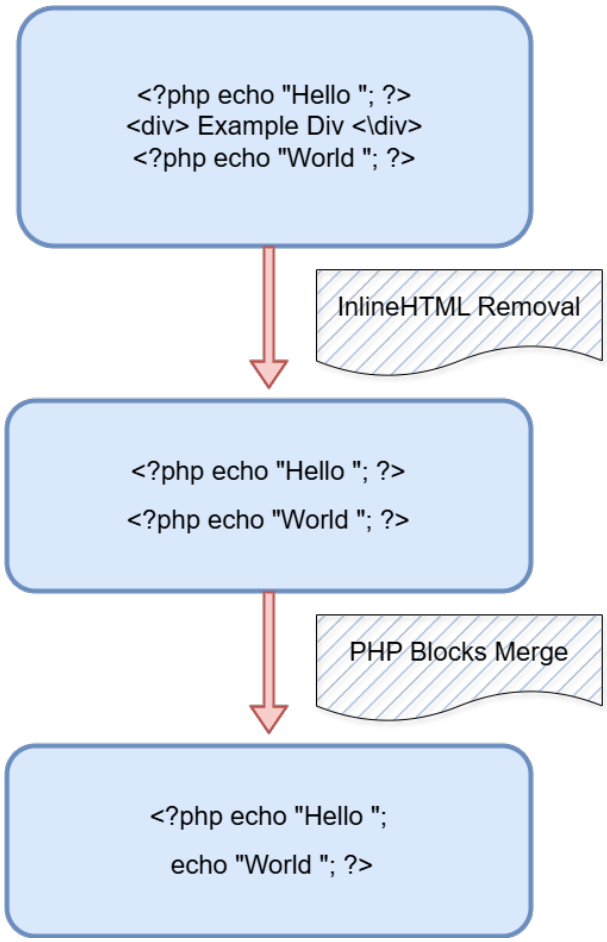

Figure 2.5 – Rule 1 workflow

code. In our approach, we will remove HTML elements represented by the node type 'InlineHTML' from the AST, focusing solely on the PHP code's behavior. This removal helps reduce the complexity of AST traversal and minimizes unnecessary computation, as HTML does not influence the PHP code's functionality.

Our process begins by parsing the AST to identify and eliminate all nodes of the 'InlineHTML' type. Consequently, the PHP file will consist of multiple PHP blocks that need to be merged into a single cohesive block, the Figure 2.5 shows the steps.

Algorithm 2.1: Rule 1: RemoveInlineHTML

```
1: function RemoveInlineHTML(originalCode)
2:     ast ← parseToAST(originalCode)
3:     for node in ast do
4:         if node is instanceof InlineHTML then
5:             ast.RemoveNode(node)
6:         end if
7:     end for
8:     refactoredCode ← astToCode(ast)
9:     MergePhpIntoOneBlock(refactoredCode) return refactoredCode
10: end function
```

— **R2. Include code from user required files**

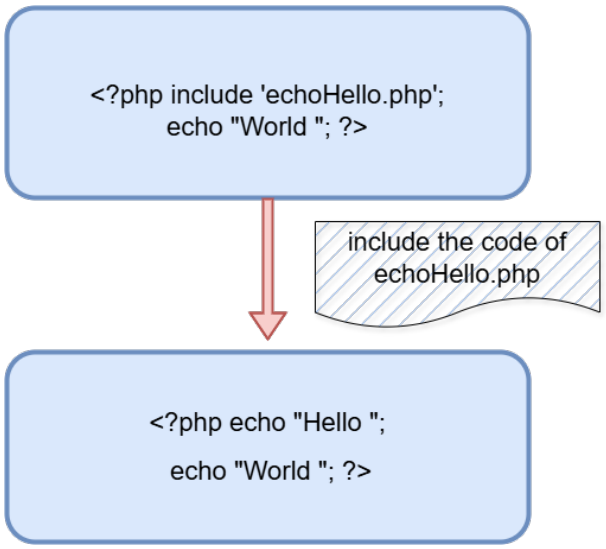

Figure 2.6 – Rule 2 workflow

In most cases Cyber-Criminals often utilize advanced techniques to obscure their malicious intent. One Common Method is deviding their code into multiple files then utilising the include and require to call the other parts of code which makes it confusing for the detection systems.

To effectively address this evasion strategy, a proactive mitigation approach has been developed, our rule as shown in Figure 2.6. This rule entails replacing all user-defined files referenced in include and require statements with their actual code before conducting thorough analysis. By preemptively resolving these dependencies, the entire code structure can be comprehensively examined, thereby reducing the effectiveness of attempts to hide malicious content through code fragmentation. This was chosen as

the second step to collect all the code in one file before the execution of other rules that manipulate php code.

Algorithm 2.2: Rule 2: Include code from user required and included files

```
1: function IncludeCode(originalCode)
2:     ast ← parseToAST(originalCode)
3:     for Each node in ast do
4:         if node is instanceof Include or node is instanceof Require then
5:             includedCode ← getcontent(node.arg)
6:             newCode ← Merge(includedCode, originalCode)
7:         end if
8:     end forreturn newCode
9: end function
```

— **R3. Replace constants by their actual values**

Refactoring for code clarification through strategies such as substituting constants

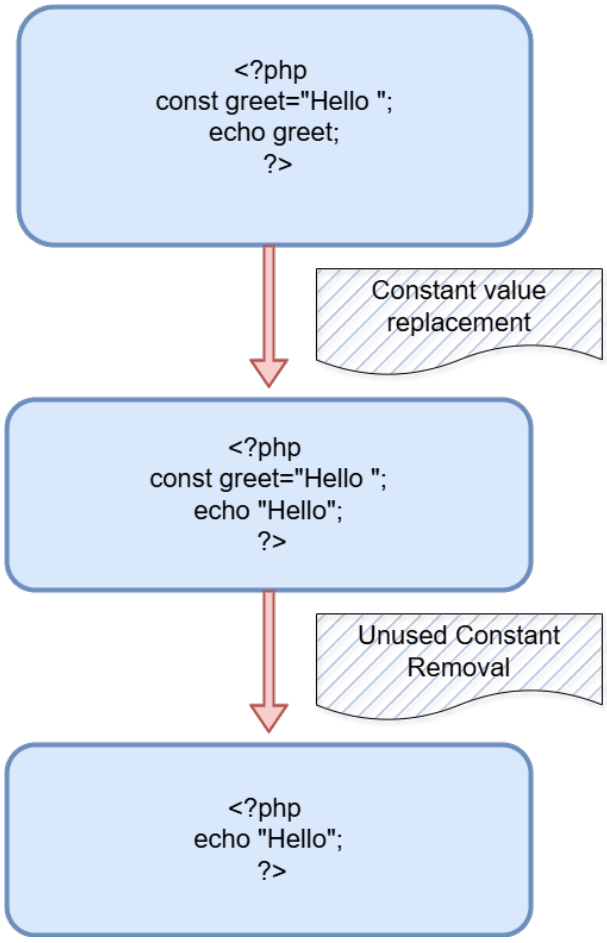

Figure 2.7 – Rule 3 workflow

with their literal values offers several significant benefits.

Firstly, simplification is a key advantage. By replacing constants with their actual values, the code becomes more transparent and easier to follow. This practice eliminates the need for repeated look-ups of constant values during execution, thereby enhancing readability and comprehension for developers.

Secondly, efficiency is notably improved. This refactoring approach can streamline operations within the codebase by removing the need to dynamically fetch constant values. Consequently, computational overhead is reduced, which leads to enhanced overall performance and responsiveness of the application.

Additionally, refactoring aids in managing code size effectively. Eliminating constant definitions helps minimize the code's footprint, simplifying maintenance tasks and optimizing resource utilization throughout the application's lifecycle.

From a security perspective, this refactoring rule is particularly relevant in PHP environments where malware detection is a concern. Malicious actors frequently use obfuscation techniques, such as employing constants to obscure harmful payloads. By substituting these constants with their actual values, the code becomes less obfuscated, thereby improving the efficacy of machine learning models designed for malware detection. This transformation facilitates clearer pattern recognition and more effective identification of suspicious behaviors or payloads.

In conclusion, adopting refactoring strategies like replacing constants with literal values not only enhances code clarity, efficiency, and manageability but also strengthens application security, especially when integrating advanced detection techniques such as machine learning for PHP malware detection. These benefits collectively contribute to maintaining robust, comprehensible, and resilient software systems.

This refactoring process involves identifying constants in the PHP code, substituting all instances of these constants with their actual values, and subsequently removing the constant definitions from the code.

Algorithm 2.3: Rule 3: Replacing constants with their actual values

```
1: function ConstCode(originalCode)
2:     ast ← parseToAST(originalCode)
3:     for Each node in ast do
4:         if node is instanceof Const then
5:             ConstName ← node.name
6:             ConstValue ← node.value
7:             ReplaceConstByValue(ast, ConstName, ConstValue)
8:             RemoveConstDeclaration(ast, ConstName)
9:         end if
10:        if node is instanceof FuncCall and node.name.tolower() == "define" then
11:            ConstName ← node.arg[0]
12:            ConstValue ← node.arg[1]
13:            ReplaceConstByValue(ast, ConstName, ConstValue)
14:            RemoveConstDefinition(ast, ConstName)
15:        end if
16:    end for
17:    newCode ← AstToCode(ast) return newCode
18: end function
```

— **R4. Handle Binary operations and variable values**

Many Cyber Criminals use Binary-OP to obfuscate their code as shown in the Figure 2.8, such a method make the malware hard to detect since the values specially strings are separated through multiple concatenations, also replacing variable values in allowed embedding only for example variables shouldn't be replaced by their real value inside of functions,if/else statements, class definitions. Below is a pseudo-code of how this rule works, here are the embedded functions :

-Function CollectValues(ast[],value[]*) which checks for assignments, if there are available assignments it checks if it is safe or not, if it is safe it checks if the right side of the assignment is a Scalar value it then it stores the value of the assigned variable in "value['assignedVariableName']",if a Variable is found in unallowed assignment it is removed from the array.

-Function ReplaceVar(ast[],value[]*) it replaces variable by it's value only in allowed locations that wouldn't change the code's functionality.

-Function removeBinaryOP(ast[]) when it finds a BinaryOP where both left and right side are Scalar values it executes the BinaryOP in a virtual environment and then replaces it's value in the code)

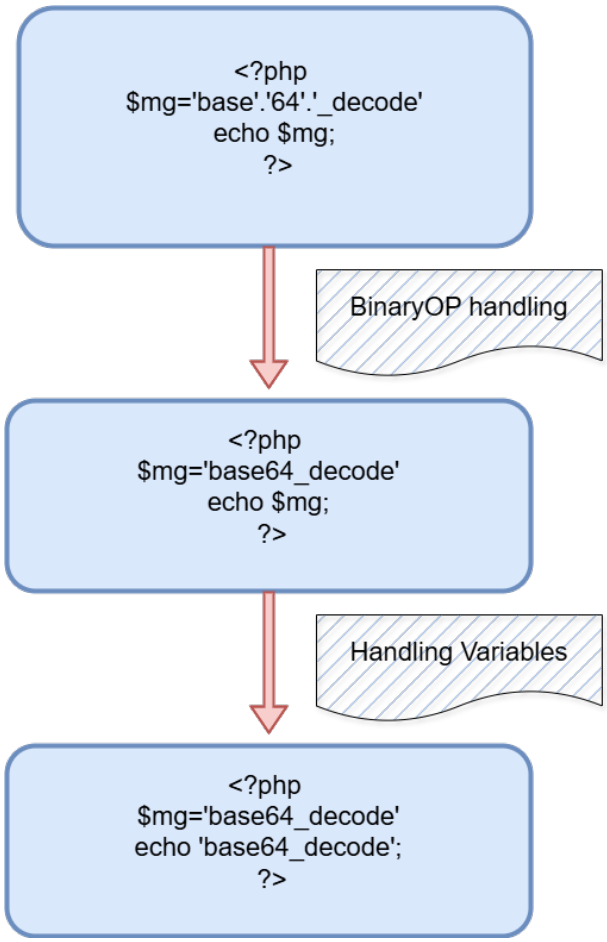

Figure 2.8 – Rule 4 workflow

Algorithm 2.4: Rule 4: Handling binaryOP

```
1: function BinaryOP(originalCode)
2:     ast ← parseToAST(originalCode)
3:     values[]                                  ▷ Initialize the values array
4:     CollectValues(ast, $value)
5:     ReplaceVar(ast, $value)
6:     removeBinaryOP(ast)
7:     newCode ← AstToCode(ast) return newCode
8: end function
```

— R5. Deobfuscation by the execution of clean functions

When it comes to detecting PHP malware, a step, in the process is to deobfuscate by running functions. Attackers often use obfuscation techniques like encoding or encrypting payloads to hide their codes purpose making it harder for static analysis and detection efforts.

The deobfuscation process involves executing decryption or decoding functions within

the obfuscated code in an environment. This helps reveal the human code without risking any harmful actions. This step is crucial for extracting features, from malware that can then be used for analysis and detection using machine learning algorithms.

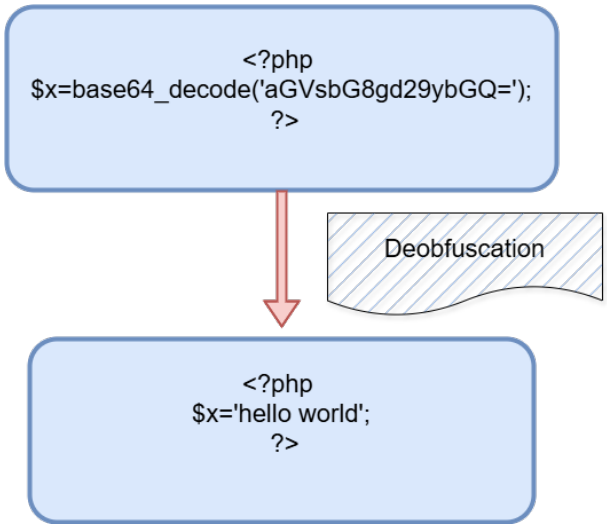

Figure 2.9 – Rule 5 workflow

By identifying and executing decryption functions, we can convert obfuscated malware into a more analyzable format. This transformation enhances our understanding of the malware's behavior and facilitates the extraction of features necessary for machine learning-based detection. Deobfuscation thus plays a crucial role in the process of detecting PHP malware, particularly when combined with refactoring and feature extraction for machine learning.

In this phase, we will establish an array of predefined functions designed for various tasks, including array manipulation, string handling, and encryption/decryption operations. Additionally, we will review any custom functions to ensure their safety and functionality by verifying that they do not perform harmful operations. Once these functions are confirmed to be secure, we will integrate them into our environment and incorporate them into our workflow.

Array Manipulation Functions:

- `array_merge()`: Merges one or more arrays.
- `array_filter()`: Filters elements of an array using a callback function.

String Manipulation Functions:

- `str_replace()`: Replaces all occurrences of the search string with the replacement string.
- `substr()`: Returns a part of a string.

Encryption/Decryption Functions:

— `base64_decode()`: Decodes data encoded with MIME base64.

— `openssl_decrypt()`: Decrypts data using the OpenSSL library.

For user-defined functions, we will inspect the code to ensure they do not call any dangerous functions or perform unsafe operations (such as eval,shell_exec..etc). If they pass this verification, we will safely execute them within our controlled environment. This process ensures that only safe and clean functions are executed, facilitating the deobfuscation and analysis of PHP malware.

Algorithm 2.5: R5. Deobfuscation by the execution of clean functions

```
1: function DeobfuscationFunc(originalCode)
2:     ast ← parseToAST(originalCode)
3:     allowedFunctionsArray[] ← ['base64_decode', 'str_replace', 'openssl_decrypt()'...]
4:     allowedFunctionArray ← allowedFunctionArray + CollectCleanUserDefinedFunctions(ast)
5:     for Each Node in ast do
6:         if Node is instanceof FuncCall and Node.name in allowedFunctionArray then
7:             code ← AstToCode(Node[])
8:             Node ← execute(code)
9:         end if
10:    end for
11:    newCode ← AstToCode(ast) return newCode
12: end function
```

— **R6. Replace variables as function call names by their values**

In the field of PHP malware detection and analysis, a notable challenge stems from obfuscated code that employs dynamic function calls through variables storing function names. This obfuscation technique is frequently utilized by malicious actors to evade detection and hinder the understanding of harmful code by security analysts and automated systems. By assigning function names to variables and subsequently invoking these variables as functions, attackers obscure the true intent of their code. This complexity complicates comprehensive static analysis, as it becomes challenging to discern the code's actual functionality and detect malicious behavior effectively.

To address this challenge, Rule R6 focuses on replacing variables used for function calls with their actual string values. This refactoring step aims to improve the clarity and comprehensibility of the code, thereby simplifying the analysis of potentially harmful PHP scripts. By substituting these dynamic function calls with their literal values, analysts can gain insight into the specific functions being executed, which is essential for identifying hidden malicious behavior.

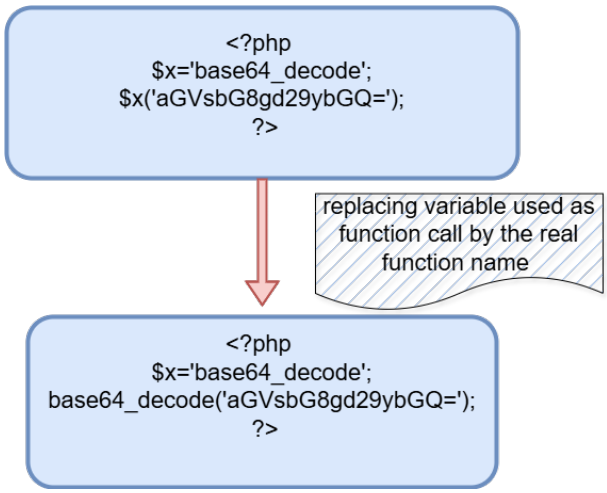

Figure 2.10 – Rule 6 workflow

Malicious actors exploit dynamic function calls through variables to dynamically alter their malware's behavior during runtime, thereby complicating traditional signature-based detection methods. This technique allows them to modify the functionality of their code without changing its core logic, increasing the difficulty of detection. The application of Rule R6 allows security analysts to dismantle these obfuscation layers, revealing the true nature of the code and facilitating more effective detection and mitigation strategies.

Furthermore, this rule enhances the extraction of accurate features needed for machine learning-based malware detection systems. By transforming obfuscated code into a structured and understandable format, analysts can extract meaningful features that improve the precision of detection models. This approach strengthens cybersecurity defenses against PHP malware, enabling proactive identification and mitigation of threats in complex and evolving environments.

In summary, Rule R6 plays a crucial role in the deobfuscation process, enabling security analysts to uncover and understand the obscured functionality of PHP malware. By replacing variables used as function call names with their actual values, analysts can enhance their ability to detect and mitigate malicious activities and aid in feature extraction, thereby reinforcing cybersecurity efforts against emerging threats in PHP ecosystems.

Our approach involves initially collecting variable values into an array during the first traversal of the Abstract Syntax Tree (AST). We then traverse the AST again, replacing any function call whose name corresponds to a variable with the variable's value from the array, if available.

Algorithm 2.6: R6. Replace variables as function call names by their values

```
1: function ReplaceVarFuncCall(originalCode)
2:     ast ← parseToAST(originalCode)
3:     variableValues ← {}
4:     for Each Node in ast do
5:         if Node is instanceof Assignment then
6:             variableName ← Node.name
7:             variableValue ← Node.value
8:             variableValues[variableName] ← variableValue
9:         end if
10:    end for
11:    for each Node in ast do
12:        if Node is instanceof FuncCall and Node.name in variableValues then
13:            Node.name ← variableValues[Node.name]
14:        end if
15:    end for
16:    newCode ← AstToCode(ast) return newCode
17: end function
```

— **R7. Replace eval() by the actual code to be executed**

Rule R7 addresses the security risks associated with the use of the eval() function in PHP. The eval() function allows for the execution of dynamically generated code strings, which introduces significant vulnerabilities, such as code injection attacks. This rule aims to enhance both security and code clarity by replacing instances of eval() with the actual code snippets they execute.

The first step of Rule R7 involves parsing the content within eval() to ensure it is a syntactically correct and legitimate PHP code string. If the content is validated as such, indicating that it can be executed safely, eval() is replaced with the parsed code content. This transformation removes the potential security risks associated with dynamic code execution, as it eliminates the need for eval() and directly incorporates the executed code into the script.

By applying Rule R7, we reduce the risk of malicious exploitation and improve the comprehensibility of the code. The removal of eval() facilitates more straightforward code analysis and security assessments, making it easier to identify and address potential vulnerabilities. Furthermore, this approach contributes to more accurate feature extraction for machine learning-based malware detection, as the code is presented in a more transparent and static format, allowing for better analysis and pattern recognition.

In summary, Rule R7 plays a crucial role in securing PHP code by substituting eval()

with its actual code content, thereby mitigating security risks and enhancing code clarity. This refactoring step is essential for maintaining robust and secure PHP applications and improving the effectiveness of malware detection systems.

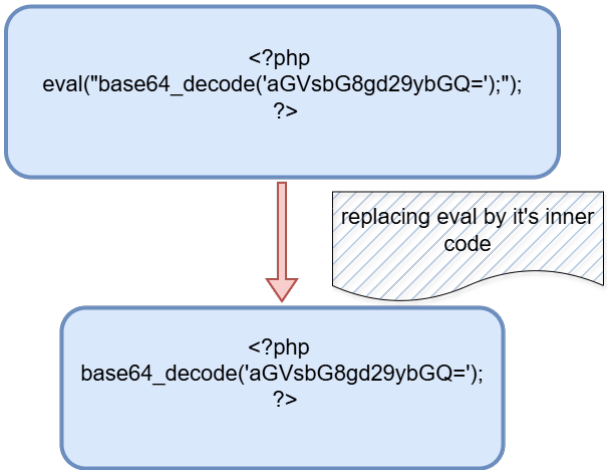

Figure 2.11 – Rule 7 workflow

By substituting ‘eval()‘ with explicit code snippets, analysts and developers achieve greater transparency into the exact operations carried out during runtime. This transparency facilitates improved comprehension and debugging of PHP scripts, thereby minimizing the potential for unintended behaviors or security vulnerabilities.

Moreover, this approach encourages safer coding practices by reducing dependence on dynamically evaluated code strings, which are challenging to audit and may introduce unforeseen risks. It also contributes to optimizing code performance by eliminating the overhead and potential errors associated with dynamic interpretation.

In summary, Rule R7 mitigates the security risks associated with ‘eval()‘ by replacing it with validated and parseable code snippets. This not only enhances code security, transparency, and performance within PHP applications but also fortifies defenses against potential vulnerabilities. Ultimately, this proactive strategy supports the development of more maintainable and secure PHP applications.

Algorithm 2.7: Rule R7: Replace eval() by the actual code to be executed

```
1: function ReplaceEval(originalCode)
2:     ast ← parseToAST(originalCode)
3:     for each node in ast do
4:         if node is instanceof Eval then
5:             evalContent ← node.content
6:             if isValidPHP(evalContent) then
7:                 parsedCode ← parse(evalContent)
8:                 replace(node, parsedCode)
9:             end if
10:        end if
11:    end for
12:    newCode ← AstToCode(ast) return newCode
13: end function
```

— **R8. Remove unused parts of code such as: variables, constants, assignments**
Rule R8 focuses on improving PHP codebases by efficiently removing unnecessary elements such as unused variables, constants, and assignments. These elements clutter the code and may obscure potentially harmful activities within PHP malware. The rule uses a structured approach to enhance code clarity and simplify the identification of security risks.

The algorithm starts by parsing the PHP code into an Abstract Syntax Tree (AST). It sets up two arrays, ‘usedVar‘ and ‘usedConst‘, to keep track of declared variables and constants. Initially, all entries in these arrays are set to ‘0‘. As it traverses the AST initially, the algorithm identifies where variables and constants are declared and marks their entries in ‘usedVar‘ and ‘usedConst‘.

During the subsequent traversal of the Abstract Syntax Tree (AST), the algorithm evaluates the usage of variables and constants in the code, excluding their initial declarations, assignments, and certain other contexts. If variables or constants are detected in these contexts, the algorithm updates the corresponding entry in usedVar or usedConst to 1.

Following the marking of usage, a second pass through the AST removes any variables or constants that still have their corresponding entry set to 0 in usedVar or usedConst. This step effectively cleans the code by eliminating unused portions, thereby enhancing code clarity and reducing unnecessary complexity.

Additionally, the algorithm addresses redundant assignments by identifying cases where a variable is assigned multiple times consecutively without any intervening usage. This optimization further streamlines the PHP code, making it more efficient and easier to maintain.

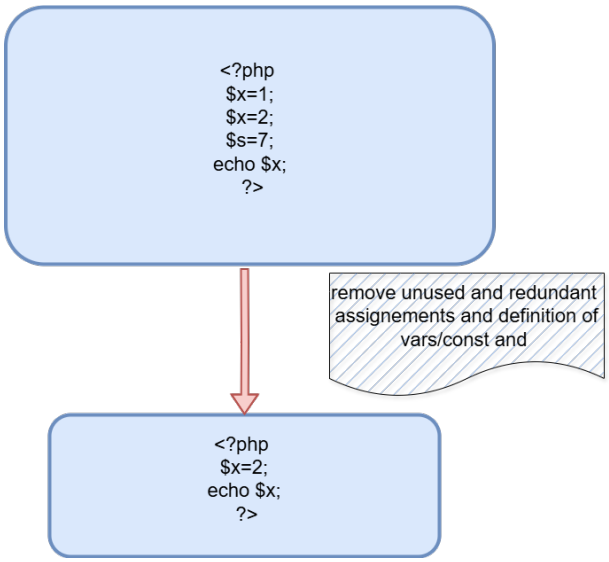

Figure 2.12 – Rule 8 workflow

In the context of PHP malware detection, Rule R8 provides significant advantages. By removing unused code segments, analysts gain clearer insights into the functioning of PHP scripts. This clarity is essential for distinguishing between legitimate functionalities and potentially harmful code patterns that may be concealed within the script.

Eliminating unused code also simplifies the analysis of PHP malware, which often includes dormant or redundant segments designed to evade detection. By concentrating analysis on essential code components, Rule R8 enhances the effectiveness of detection algorithms and automated tools, facilitating the prompt identification of suspicious behaviors indicative of malware presence.

Furthermore, optimizing PHP code through Rule R8 promotes efficient resource usage and fortifies cybersecurity measures. It supports proactive identification and mitigation of emerging threats within PHP environments, contributing to a more secure development and operational framework.

In summary, Rule R8 is instrumental in PHP malware detection by refining code structures and focusing analysis efforts. This approach not only strengthens defenses against sophisticated threats but also fosters robust cybersecurity practices within PHP ecosystems.

Algorithm 2.8: Rule R8: Remove unused parts of code

```
1: function RemoveUnused(originalCode)
2:     ast ← parseToAST(originalCode)
3:     usedVar ← empty list
4:     usedConst ← empty list
5:     for each node in ast do
6:         if node is instanceof VarDeclaration then
7:             usedVar[node.name] ← 0
8:         end if
9:         if node is instanceof ConstDeclaration then
10:            usedConst[node.name] ← 0
11:        end if
12:    end for
13:    for each node in ast do
14:        if node is usage of Var or Const and not in definitions and declarations then
15:            if node is Var then
16:                usedVar[node.name] ← 1
17:            end if
18:            if node is Const then
19:                usedConst[node.name] ← 1
20:            end if
21:        end if
22:    end for
23:    for each node in ast do
24:        if node is VarAssignment then
25:            varName ← node.name
26:            if usedVar[varName] == 0 then
27:                remove(node)
28:            end if
29:        end if
30:    end for
31:    for each node in ast do
32:        if node is redundant assignment then
33:            varName ← node.name
34:            if usedVar[varName] == 0 then
35:                remove(node)
36:            end if
37:        end if
38:    end for
39:    newCode ← AstToCode(ast) return newCode
40: end function
```

— **R9. Normalize the variables and user-defined function names**

Rule R9 is all about bringing clarity and consistency to PHP code. It does this by normalizing the names of variables and user-defined functions throughout the code. The process starts with the algorithm converting the PHP code into an Abstract Syntax Tree (AST). This tree represents the structure of the code in a way thats easy to work with.

During its first journey through the AST, the algorithm spots user-defined functions and stores their names in an array called userDefinedFunctions. Each time it finds a user-defined function, it replaces the functions name with a standardized identifier, function.

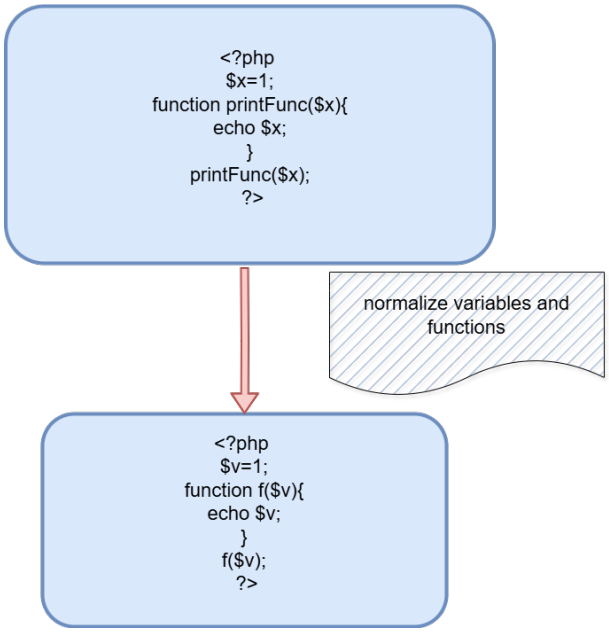

Figure 2.13 – Rule 9 workflow

During its subsequent traversal of the Abstract Syntax Tree (AST), the algorithm examines each node to identify function calls and variable declarations. When a function call is identified as a user-defined function (as indicated by an entry in userDefinedFunctions), the algorithm replaces the function call name with 'f'. Similarly, variable declaration names are replaced with 'v', ensuring consistency throughout the codebase.

After applying these changes, the algorithm reconstructs the modified AST back into PHP code, reflecting the standardized variable and function names in the final script.

In the realm of PHP malware detection and analysis, Rule R9 is pivotal. By standardizing variable and function names, it enhances the readability and comprehensibility of

the code. This improved clarity facilitates the analysis of the code and the identification of potential security risks or suspicious behaviors within PHP malware.

Furthermore, normalizing variable and function names contributes to better code maintenance practices. It mitigates ambiguity and simplifies collaboration among developers. By fostering uniformity within PHP codebases, Rule R9 supports more secure and efficient development practices, resulting in higher quality and more maintainable code.

In summary, Rule R9s standardization process enhances the clarity and understandability of PHP code. It not only aids in the detection and analysis of PHP malware but also promotes superior code quality and maintainability, thereby supporting secure and efficient development practices within PHP environments.

Algorithm 2.9: Rule R9: Normalize variables and user-defined function names

```
1: function NormalizeNames(originalCode)
2:     ast ← parseToAST(originalCode)
3:     userDefinedFunctions ← empty list
4:     for each node in ast do
5:         if node is instanceof UserDefinedFunction then
6:             userDefinedFunctions.append(node.name)
7:             node.name ← 'function'
8:         end if
9:     end for
10:    for each node in ast do
11:        if node is instanceof FuncCall then
12:            if node.name in userDefinedFunctions then
13:                node.name ← 'f'
14:            end if
15:        end if
16:        if node is instanceof VarDeclaration or node is instanceof Variable then
17:            node.name ← 'v'
18:        end if
19:    end for
20:    newCode ← AstToCode(ast) return newCode
21: end function
```

Rules Management

Having such rules needs good management to minimize the iteration numbers, we have ordered the rules in the best order possible after some experiments.

When starting the execution of our system with Rule 1 until Rule 9 , we start by generating an MD5 hash of the file and each time a refactoring rule is applied to the file we generate a new MD5 hash and compare it to the original one to check if the file has changed after applying the rule or not if not the system proceeds to the next rule else when the file has changed the system will move back to the first Rule and so on, the system stops when the file hasn't changed at all through all the rules or if an MD5 hash was repeated through the process to avoid infinite processing recursively . This principle doesn't apply to Rule 9, when a file reaches Rule 9 the system terminates and moves to the next file.

2.2.3 Feature extraction

Our experiment with traditional machine learning classifiers for webshell detection hinges on a comprehensive feature set identified through a prior systematic mapping study [28]. This study meticulously examined existing research and identified a total of 98 unique features that can be instrumental in detecting malicious PHP webshells. These features are meticulously categorized into three distinct groups, each serving a specific purpose in uncovering webshell activity.

Statistical Features: Unveiling Obfuscated Deception Statistical features serve as detectives for obfuscated and encrypted content embedded within source code files. Threat actors, in their attempts to mask the malicious intent of their scripts, frequently resort to obfuscation and encryption techniques. Statistical features act as a shield against these tactics by analyzing characteristics of the code that can hint at manipulation. Here are some prominent examples of statistical features employed for webshell detection:

— Information Entropy: This metric quantifies the level of randomness or uncertainty within the code. High entropy can indicate the presence of obfuscation.

— Index of Coincidence: This feature assesses the frequency of recurring character patterns within the code.

— File Compression Ratio: This feature compares the size of the original code file to its compressed version.

Syntactical Features: Exposing Malicious Functionality Syntactical features focus on capturing the structure and flow of the code, specifically targeting the presence of malicious expressions often used by attackers. These expressions can be employed to steal sensitive information or manipulate system configurations. By scrutinizing the code's syntax, the machine learning model can learn to identify red flags associated with malicious intent. Here are some key examples of syntactical features:

— Number of Loops and Conditional Statements

Lexical Features: Recognizing Malicious Text Patterns Lexical features delve into the world of text patterns and strings embedded within the code and comments. These patterns can provide valuable clues about the code's functionality. By analyzing these lexical elements, the machine learning model can learn to identify indicators of malicious intent. Here are some commonly used lexical features:

— Number of Calls to Specific Functions (e.g., eval(), base64_decode())

— Occurrence of Specific Commands (e.g., gcc, chmod)

— Utilization of Specific Variables (e.g., $_POST, $_GET, $_CMD)

By incorporating these diverse feature sets, traditional machine learning models can be trained to effectively detect webshells hidden within PHP code. Each feature category plays a crucial role in uncovering the deceptive tactics employed by attackers.[27]

For our DNN model we have chosen TF-IDF for feature extraction. TF-IDF[16] is a widely used feature representation in text mining, combining the frequency of a word in a document (TF) and the inverse document frequency (IDF) across a collection of documents. The term frequency (TF) is calculated as shown in Equation 2.1, the inverse document frequency (IDF) is calculated as shown in Equation 2.2, and the TF-IDF score is the product of TF and IDF, as shown in Equation 2.3. TF-IDF has shown

$$\mathrm{TF}(t, D) = \frac{\mathrm{count}(t, d)}{\sum_k \mathrm{count}(k, d)} \tag{2.1}$$

where $\mathrm{count}(t, d)$ denotes the number of occurrences of word t in document d, and $\sum_k \mathrm{count}(k, d)$ denotes the total number of all words in document d.

$$\mathrm{IDF}(t, D) = \log \frac{N}{\mathrm{df}(t, D) + 1} \tag{2.2}$$

where N denotes the total number of documents in the collection, and $\mathrm{df}(t, D)$ denotes the number of documents that contain the word t.

$$\text{TF-IDF}(t, D) = \mathrm{TF}(t, D) \times \mathrm{IDF}(t, D) \tag{2.3}$$

The TF-IDF model is highly effective in various domains such as search engines, text classification, and information retrieval, significantly enhancing text analysis capabilities. In the domain of PHP malware detection, particularly in the context of Webshell detection, TF-IDF vectorization has proven to be crucial for improving detection accuracy. In this study, abstract syntax tree sequences obtained in the preceding steps are subjected to vectorization and feature extraction using TF-IDF. TF-IDF proved it's efficacity in recent work done for detecting webshells using AST-DF model and TF-IDF for feature extraction with an accuracy of 99.61%.[20]

2.3 Conclusion

In this chapter, we have delved into the pervasive threat posed by PHP malware within web applications, exacerbated by cybercriminals adeptly evading traditional detection methods through obfuscation techniques. PHP, as the predominant language for web development, amplifies the challenge of securing applications against evolving cyber threats. Obfuscation techniques like encryption, encoding, and code fragmentation are extensively used to disguise malicious payloads, complicating detection for security systems and machine learning models alike. The presence of legitimate uses of obfuscation, such as code protection against intellectual property theft, blurs the line between benign and malicious intent. Recognizing these complexities, our approach advocates PHP code refactoring as a strategic countermeasure to enhance malware detection accuracy. Through static analysis and manipulation of Abstract Syntax Trees (AST), our method aims to improve code readability without altering functionality. This refactoring process not only helps mitigate obfuscation but also enables clearer feature extraction for machine learning models, thereby enhancing their ability to differentiate between benign and malicious PHP code. The proposed system architecture integrates AST generation, code refactoring, and feature extraction as foundational steps critical for preparing datasets conducive to training robust machine learning models. These models are essential for identifying subtle anomalies indicative of malware. Furthermore, our refactoring rulessuch as eliminating unnecessary HTML, resolving include dependencies, substituting constants with their values, and managing complex binary operationsare designed to systematically enhance code clarity and streamline detection efforts. Each rule addresses specific challenges posed by obfuscated PHP malware, ensuring that refactored code remains both understandable to humans and interpretable by machines.

Looking ahead, the effectiveness of our approach hinges on continuous refinement and adaptation to emerging obfuscation tactics and malware variants. Collaboration among cybersecurity experts, software developers, and researchers is crucial to staying ahead of evolving threats and effectively safeguarding PHP-based web applications. In conclusion, as PHP continues to dominate web development, proactive measures like code refactoring represent a promising approach to fortifying cybersecurity defenses. By advancing the state-of-the-art in malware detection through improved code readability and precise feature extraction, we contribute to a more secure digital landscape resilient against sophisticated cyber threats.

Chapter 3

Implementation and Result

This chapter discusses the empirical results obtained from applying various machine learning algorithms to the task of detecting PHP malware before and after code refactoring.

3.1 Dataset

In this study, we utilized a Dataset that is generated from a group of php files through feature extraction, the first group of files was provided by the supervisor and used previously in research that he was a member of [27].
The Dataset contain Benign PHP files and Malicious PHP files that are webshells.

Dataset	Benign Files	Malicious Files	Total Rows
Dataset 1	1081	1333	2414

Table 3.1 – Summary of Datasets

3.1.1 Dataset Features:

Below is a detailed description of each feature of the dataset we are using that were extracted following the method developed by A.Hannousse,Nait-Hamoud,M. C., AND Yahiouche [26]:

- **filename**: The name of the PHP file being analyzed.
- **hash**: Cryptographic hash of the file, used for verifying file integrity.
- **sourcecode**: The source code of the PHP file.
- **opcode**: The opcode sequence generated from the PHP file.
- **LanguageIC**: Indicator of the programming language version or type.
- **Entropy**: Measures the randomness or disorder in the file.

— **LongestWord**: The longest word in the PHP source code.

— **SignatureNasty**: Identifies known malicious signatures in the file.

— **Compression_ratio**: The compression ratio of the file, comparing compressed and uncompressed size.

— **size**: The size of the file in bytes.

— **#var**: The number of variables used in the file.

— **#global**: The number of global variables declared.

— **#if**: The number of ‘if‘ statements in the source code.

— **#loop**: The number of loop constructs such as ‘for‘ or ‘while‘.

— **#str**: The number of string operations present in the file.

— **#neq_str**: The count of string inequality operations.

— **#def_func**: The number of defined functions in the code.

— **#eval**: The count of ‘eval()‘ function usage, which can be indicative of malicious intent.

— **#isinstanceof**: The count of ‘instanceof‘ operator usage.

— **#htmlspecialchars**: The number of ‘htmlspecialchars()‘ function calls, used for sanitizing output.

— **#concat**: The number of string concatenations.

— **#special_char**: The number of special characters used in the file.

— **#request**: The count of ‘$_REQUEST‘ superglobal usage.

— **#post**: The count of ‘$_POST‘ superglobal usage.

— **#get**: The count of ‘$_GET‘ superglobal usage.

— **#server**: The count of ‘$_SERVER‘ superglobal usage.

— **#cookie**: The count of ‘$_COOKIE‘ superglobal usage.

— **#session**: The count of ‘$_SESSION‘ superglobal usage.

— **#files**: The count of ‘$_FILES‘ superglobal usage.

— **#globals**: The count of ‘$_GLOBALS‘ superglobal usage.

— **#no_reg_domain**: Occurrences of non-registered domain references.

— **#pcntl_exec**: The count of ‘pcntl_exec()‘ calls.

— **#str_rot13**: The count of ‘str_rot13()‘ usage, often used for basic encoding.

— **#json_decode**: The count of ‘json_decode()‘ usage.

— **#base64_decode**: The count of ‘base64_decode()‘ usage, which is often used in obfuscation.

— **#gzinflate**: The count of ‘gzinflate()‘ usage, typically used in compression or obfuscation.

— **#exe**: The count of ‘.exe‘ file references in the PHP code.

— **#gzuncompress**: The count of ‘gzuncompress()‘ function usage.

— **#file_put_contents**: The number of times ‘file_put_contents()‘ is used to write to files.

— **#fwrite**: The count of ‘fwrite()‘ function usage for file writing.

— **#chmod**: The count of ‘chmod()‘ function usage for changing file permissions.

— **#str_replace**: The count of ‘str_replace()‘ function calls.

— **#system**: The count of ‘system()‘ function usage, often associated with command execution.

— **#preg_replace**: The count of ‘preg_replace()‘ function usage.

— **#create_function**: The number of dynamically created functions using ‘create_function()‘.

— **#pack**: The count of ‘pack()‘ function usage for binary data.

— **#implode**: The count of ‘implode()‘ function calls to join array elements into a string.

— **#explode**: The count of ‘explode()‘ function calls to split a string into an array.

— **#exec**: The number of ‘exec()‘ function calls for command execution.

— **#getenv**: The count of ‘getenv()‘ calls, retrieving environment variables.

— **#mail**: The number of ‘mail()‘ function calls.

— **#rand**: The number of ‘rand()‘ function calls, generating random numbers.

— **#chr**: The number of ‘chr()‘ function calls.

— **#uname**: The number of ‘uname()‘ calls, which fetches system information.

— **#php_uname**: The number of ‘php_uname()‘ calls for system information.

— **#php_os**: The number of ‘php_os‘ references, representing the operating system.

— **#http_build_query**: The number of ‘http_build_query()‘ function calls.

— **#is_readable**: The number of ‘is_readable()‘ function calls, checking file readability.

— **#is_writable**: The number of ‘is_writable()‘ function calls, checking file writability.

— **#chdir**: The number of ‘chdir()‘ calls for changing directories.

— **#scandir**: The number of ‘scandir()‘ function calls.

— **#rmdir**: The number of ‘rmdir()‘ function calls for removing directories.

— **#getcwd**: The number of ‘getcwd()‘ function calls, retrieving the current working directory.

— **#disk_free_space**: The number of ‘disk_free_space()‘ function calls.

— **#disk_total_space**: The number of ‘disk_total_space()‘ function calls.

— **#sys_get_temp_dir**: The number of ‘sys_get_temp_dir()‘ calls to retrieve the system’s temporary directory.

— **#passthru**: The count of ‘passthru()‘ function usage, often linked to shell command execution.

— **#shell_exec**: The number of ‘shell_exec()‘ calls, which execute shell commands.

— **#shell**: The number of ‘shell‘ function calls.

— **#pcntl**: The count of ‘pcntl‘ (Process Control) function usage.

— **#popen**: The count of ‘popen()‘ function usage, often used for process control.

— **#proc_open**: The count of ‘proc_open()‘ function usage for opening processes.

— **#python_eval**: The count of ‘eval()‘ usage in Python contexts within the code.

— **#show_source**: The number of ‘show_source()‘ calls, revealing source code.

— **#touch**: The number of ‘touch()‘ function calls to modify file timestamps.

— **#preg_match**: The number of ‘preg_match()‘ calls for regular expression matching.

— **#unlink**: The count of ‘unlink()‘ calls, used to delete files.

— **#md5**: The number of ‘md5()‘ hash function calls.

— **#ini_get**: The number of ‘ini_get()‘ function calls, retrieving PHP configuration settings.

— **#phpinfo**: The number of ‘phpinfo()‘ calls, which reveal the PHP environment information.

— **#phpversion**: The number of ‘phpversion()‘ calls to retrieve the PHP version.

— **#define**: The number of ‘define()‘ calls, which define constants.

— **#header**: The count of ‘header()‘ function calls to send HTTP headers.

— **#extract**: The number of ‘extract()‘ calls, extracting variables from arrays.

— **#mysql_query**: The number of MySQL query function calls.

— **#file_get_contents**: The number of ‘file_get_contents()‘ calls for reading file contents.

— **#rename**: The count of ‘rename()‘ function usage, renaming files or directories.

— **#assert**: The number of ‘assert()‘ calls, often misused in attacks.

— **#serialize**: The number of ‘serialize()‘ calls to serialize data.

— **#unserialize**: The number of ‘unserialize()‘ calls to unserialize data.
— **#openssl_decrypt**: The number of ‘openssl_decrypt()‘ calls for decrypting data.
— **#mt_rand**: The number of ‘mt_rand()‘ function calls.
— **#fread**: The number of ‘fread()‘ calls for reading file content.
— **#fget**: The number of ‘fget()‘ calls for file reading.
— **#fopen**: The number of ‘fopen()‘ calls for opening files.
— **#setcookie**: The number of ‘setcookie()‘ function calls.
— **#move_uploaded_file**: The number of ‘move_uploaded_file()‘ calls for file uploads.
— **#required_files**: The number of required files in the script.
— **#included_files**: The number of included files in the script.
— **#used_files**: The number of files used by the script.
— **max_len_js_file_name**: Maximum length of JavaScript file names referenced in the PHP file.
— **#mixed_case_str**: The count of mixed-case string usage.
— **#funcc_as_var**: The number of function calls using a variable.
— **webshell_by**, **hack_by**, **bypass_AV**, **password_is**: Occurrences of these strings in comments or code, often indicative of webshell or hacking attempts.
— **label**: The label indicating whether the file is benign or malicious.

3.1.2 Information Gain

Information Gain is defined as follows:

$$IG(D, A) = H(D) - H(D|A) \tag{3.1}$$

where:

— $H(D)$ is the entropy of the dataset D, which quantifies the impurity or uncertainty in the data.
— $H(D|A)$ is the conditional entropy of D given feature A, representing the uncertainty remaining after splitting the data based on feature A.

3.1.2.1 Feature Selection Using Mutual Information

The `mutual_info_classif` function from `scikit-learn` can be used to compute mutual information, which is conceptually similar to Information Gain. This function helps in selecting the most relevant features by measuring the dependency between each feature and the target variable (malware or benign).

3.1.2.2 Visualization

Once Information Gain is calculated, visualization tools such as Seaborn and Matplotlib can be used to present the results. For instance, a bar plot of the top features and their information gain scores can reveal which PHP code patterns are most associated with malware.

Using Information Gain for feature selection in PHP malware detection allows for the identification of the most informative features. By focusing on key indicators, such as specific PHP functions or obfuscation techniques, more effective detection models can be developed. The combination of `mutual_info_classif` and visualization libraries enhances both the feature selection process and the interpretability of the detection system.

3.1.2.3 Results of Information Gain Analysis

In this section, we'll discuss the outcomes of our feature selection process using Information Gain on two datasets. Information Gain is a key technique for evaluating which features are the most useful for predicting outcomes. It measures how much knowing the value of a feature reduces uncertainty about the class label. Essentially, it tells us how much information a feature contributes to making predictions. By applying this method, we've been able to identify the most relevant features that provide significant insights into the data, helping to enhance the performance of our models.

— **Before Refactoring:**

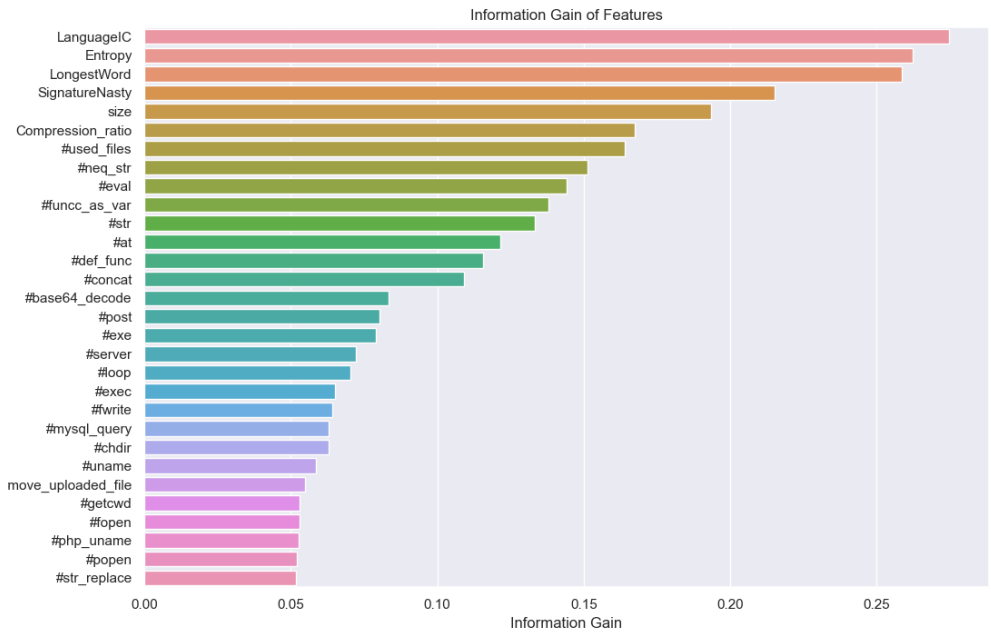

Figure 3.1 – Before Refactoring dataset

— **After Refactoring:**

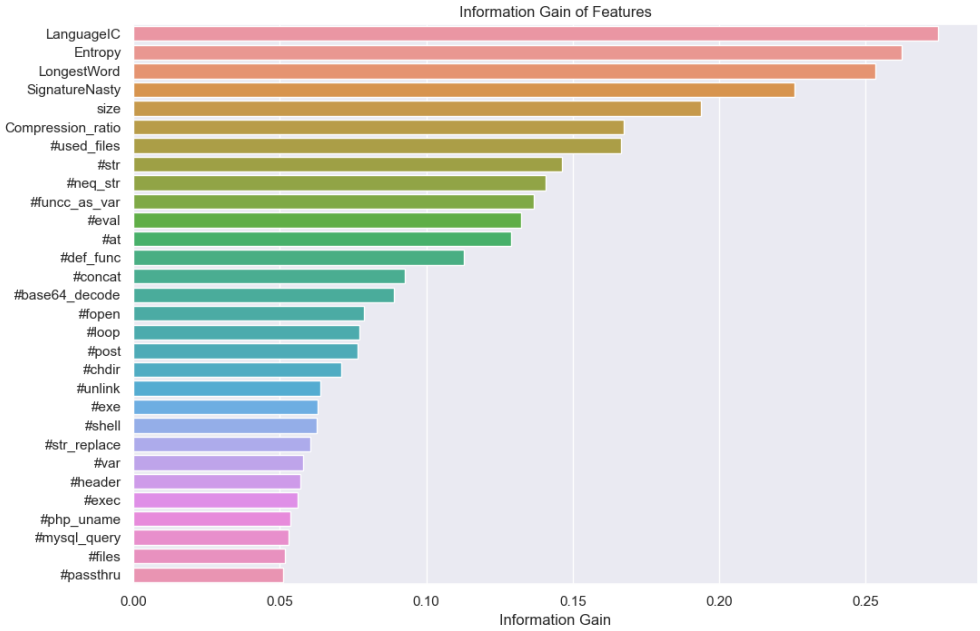

Figure 3.2 – After Refactoring dataset

The graphs 3.1 and 3.2 above illustrate the Information Gain scores for the top 30 features Before and After Refactring Dataset. The x-axis represents the feature names, while the y-axis shows the Information Gain values. The features with the highest Information Gain are positioned to the right, indicating their greater relevance for distinguishing between malicious and benign PHP scripts. The results doesn't show a great change in the importance of features, though we see some of the Operating system functions showing up to our top 30 features such as : files,unlink,shell. We notice also that some features as eval, and base64decode functions are still nearly on the same level of importance, and that goes to the fact that most of them are using POST,GET which makes it hard to eliminate them.

By computing the Information Gain for these features, we can rank them in order of importance, identifying which characteristics are most indicative of malicious files.

3.2 Validation process

3.2.1 Machine Learning Classifiers

In this section, we provide an overview of the machine learning classifiers used in our research. Each algorithm is explained with its architecture and its contribution in PHP

malware detection.

3.2.1.1 Bernoulli Naive Bayes

The Bernoulli Naive Bayes algorithm is a probabilistic classifier based on Bayes' Theorem, specifically designed for binary feature vectors. It assumes that each feature is either present or absent (true or false). This is ideal for text classification tasks, where the presence or absence of specific tokens plays a crucial role.

For PHP malware detection, Bernoulli Naive Bayes is effective when features are binary (e.g., the presence of suspicious code snippets or function calls). By calculating the probability that a PHP script is malicious based on the presence of these features, it can make rapid and accurate predictions [57].

3.2.1.2 K-Nearest Neighbors (KNN)

K-Nearest Neighbors is a non-parametric, instance-based learning algorithm that classifies new instances by comparing them to the k-nearest instances in the feature space. The classification is determined by majority voting from the nearest neighbors.

In detecting PHP malware, KNN can identify similarities between known malicious scripts and new, unseen scripts. By analyzing structural and syntactic features, KNN helps to detect malware based on proximity to known malicious patterns [58].

3.2.1.3 Decision Tree

A Decision Tree is a hierarchical, tree-like structure where internal nodes represent different decision points based on features, and the branches represent the possible outcomes of each decision. It recursively splits the dataset until each final node represents a class label.

For PHP malware detection, Decision Trees provide a clear and interpretable flow of decisions, such as identifying suspicious code behaviors or anomalous function calls. Each branch of the tree represents a series of decisions that leads to the final classification [24].

3.2.1.4 Logistic Regression

Logistic Regression is a linear model used for binary classification tasks. It estimates the probability that a given input belongs to a particular class by fitting a linear decision boundary and applying the logistic function.

In PHP malware detection, Logistic Regression helps in identifying the probability of a script being malicious by analyzing key features like control flow structures, API calls, and obfuscation patterns. The model outputs a probability score, which can then be thresholded to make a final classification [17].

3.2.1.5 Support Vector Machine (SVM)

Support Vector Machine (SVM) is a supervised learning algorithm that aims to find the optimal hyperplane that separates data points from different classes with the largest margin. It is particularly effective for high-dimensional spaces and can handle both linear and non-linear classification using kernel functions.

In the context of PHP malware detection, SVM is useful for distinguishing between benign and malicious scripts by learning from complex patterns in the code. Features such as function usage, control structures, and execution paths are mapped into a higher-dimensional space to find the best decision boundary [47].

3.2.1.6 Random Forest

Random Forest is an ensemble learning method that builds multiple decision trees during training and combines their predictions to make a more robust classification. Each tree is built from a random subset of features and data, which helps reduce overfitting and improve generalization.

In PHP malware detection, Random Forest helps by analyzing multiple features from the PHP code (such as function usage, control flow, and external API calls) and using multiple decision trees to improve the accuracy of detecting malicious behavior [37].

3.2.1.7 DNN Source Code Model

The DNN Source Code Model demonstrates near-perfect accuracy in classification tasks, making it highly effective for PHP malware detection. Its ability to maintain strong performance even after refactoring highlights its adaptability to changes in code. This model can be employed to detect intricate patterns of malicious behavior embedded in PHP source code. Its high precision and F1 score ensure that it can identify malware accurately while minimizing false positives. As a result, the DNN Source Code Model is a powerful tool for detecting evolving PHP malware, where subtle deviations in code structure may indicate malicious intent [46].

3.2.1.8 DNN op Code Model

The DNN op Code Model initially performed well, with solid accuracy and precision, but experienced a noticeable decline after refactoring, particularly in recall and F1 score. Despite this, it remains useful for PHP malware detection, especially in identifying known patterns or signatures of malicious code. However, the post-refactoring drop suggests that this model may struggle to generalize or adapt to changes in malware tactics or newly emerging threats. To be fully effective in dynamic environments like PHP, further refinement or model augmentation would be necessary to improve its adaptability to new malware variants [8].

3.2.1.9 RF-DNN2 Model

The RF-DNN2 Model is a consistently high-performing algorithm, maintaining near-perfect accuracy both before and after refactoring. Its resilience to changes in data and reliable classification performance make it an ideal candidate for PHP malware detection. The models stability ensures that it can consistently identify malicious PHP code, regardless of evolving tactics used by malware developers. With its strong precision and recall, RF-DNN2 can play a pivotal role in protecting PHP applications from malicious attacks, providing a dependable solution in both static and dynamic malware detection environments [26].

3.2.1.10 AdaCost

AdaCost is a cost-sensitive boosting algorithm designed to handle imbalanced classification problems. It assigns different weights to misclassified instances based on their importance and iteratively adjusts these weights to improve classification accuracy.

In the context of PHP malware detection, AdaCost helps address the imbalance between benign and malicious scripts, ensuring that false negatives (missed malware) are penalized more heavily than false positives. This improves the detection of hard-to-identify malicious scripts [59].

3.2.1.11 Gradient Boosting Decision Trees (GBDT)

Gradient Boosting Decision Trees (GBDT) is an ensemble learning technique that builds trees sequentially, with each tree correcting the errors made by the previous one. It optimizes a differentiable loss function using gradient descent.

For PHP malware detection, GBDT is effective in learning complex interactions between features such as code structure, external function calls, and control flow anomalies. It combines the strengths of multiple weak learners (decision trees) to achieve a highly accurate model [18].

3.2.2 Validation Metrics and Evaluation process

When developing machine learning models for PHP malware detection, it is critical to ensure that the model not only identifies malicious code effectively but also minimizes false positives and false negatives. In a security context, missclassifying malware can have serious consequences, as false negatives (failing to detect malware) may lead to compromised systems, while false positives (incorrectly identifying benign code as malware) can unnecessarily disrupt operations.

To evaluate the performance of our machine learning classifiers, we employ several validation metrics, each focusing on different aspects of the model's ability to correctly classify malicious and benign PHP code. In this section, we present the most widely used

validation metrics in the literature: Recall, Precision, F1 Score, and Accuracy. These metrics provide a comprehensive view of the classifier's performance, especially when dealing with an imbalanced dataset where malicious PHP code is rarer than benign code. The following subsections provide the formal definitions and formulas for each metric used.

3.2.2.1 Recall

Recall, also known as sensitivity or true positive rate, measures the ability of the model to correctly identify all positive instances (i.e., malware). It is particularly important in the context of PHP malware detection, where failing to detect a piece of malware could have severe security consequences. Recall is defined as follows:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

where:

— TP is the number of true positives (correctly classified malware instances),

— FN is the number of false negatives (malware instances classified as benign).

3.2.2.2 Precision

Precision measures the accuracy of positive predictions made by the model. In other words, it quantifies how many of the instances classified as malware are actually malware. Precision is particularly useful when it is important to minimize false positives, such as in PHP malware detection, where a benign piece of code should not be mistakenly flagged as malicious. Precision is defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

where:

— TP is the number of true positives,

— FP is the number of false positives (benign instances classified as malware).

3.2.2.3 F1 Score

The **F1 Score** is the harmonic mean of precision and recall, providing a balance between the two. It is particularly useful when there is an uneven class distribution, such as in malware detection, where the number of benign files far exceeds the number of malicious files. The F1 score is defined as:

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 score takes both false positives and false negatives into account, making it a more comprehensive measure of the classifiers effectiveness.

3.2.2.4 Accuracy

Accuracy is the most intuitive performance metric and measures the overall correctness of the model's predictions. While it is useful for balanced datasets, it may not be as informative for imbalanced datasets, such as in PHP malware detection, where benign code far outweighs malicious code. Accuracy is defined as:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

where:

— TP is the number of true positives,

— TN is the number of true negatives (benign instances correctly classified as benign),

— FP is the number of false positives,

— FN is the number of false negatives.

3.2.2.5 Cross-Validation

For this research, we used cross-validation to make sure our Deep Neural Network (DNN) models were both reliable and generalizable. Cross-validation is a method used to assess how well a machine learning model performs on unseen data. Specifically, we applied k-fold cross-validation.

In k-fold cross-validation, the dataset is split into k equally sized parts, or "folds." We then train the model on $k-1$ of these folds and use the remaining fold to test the model. This process is repeated k times, with each fold being used once as the validation set. Finally, we average the results from all k iterations to get a final performance score.

For our study, we chose $k = 10$. This means we divided our data into 10 parts. Each time, we trained the DNN on 9 parts and tested it on the 1 part that was left out. This approach helps to ensure that our model is not just fitting to a specific portion of the data and gives us a better idea of how it will perform on new data.

Deep Neural Networks (DNNs) is a bit complex, with many parameters that can easily lead to overfitting if not properly managed. To avoid this problem, we used k-fold cross-validation to test how well our DNN models generalized across different parts of the dataset.

Cross-validation allowed us to evaluate our DNNs' performance by training and testing them on various subsets of the data. This way, we could ensure that the models were effective in recognizing patterns and making predictions on different data samples.

Cross-validation is particularly useful when working with security problems, in our case detecting PHP malware, the dataset can be very diverse, including many different types of malware and benign activities. Using cross-validation helps ensure that our DNN models are not biased towards any specific type of malware or benign behavior. This is important for building a model that can accurately detect various types of PHP malware in real-world scenarios.

After running k-fold cross-validation, we collected and averaged the performance metrics of our DNN models, including accuracy, precision, recall, and F1 score. This process gave us a reliable measure of how well our models performed and how well they are likely to work with new, unseen data.

3.3 Results & Analysis for dataset

3.3.1 Bernouli Naive Bayes Model

Bernouli Naive Bayes Model	Before Refactoring dataset	R (%): 61.54
		P (%): 93.78
		A (%): 77.09
		F1 (%): 74.20
	After Refactoring dataset	R (%): 57.22
		P (%): 94.67
		A (%): 76.12
		F1 (%): 71.23

Table 3.2 – Bernouli Naive Bayes Results datasets

— **Before Refactoring:** It has low Recall (61.54%), meaning it misses many malicious files. However, it shows high Precision (93.78%), indicating few false positives. Accuracy (77.09%) is decent but lower compared to other models, and the F1 Score (74.20%) reflects a trade-off between Precision and Recall.

— **After Refactoring:** Recall drops to 57.22%, showing decreased ability to detect malicious files. Precision remains high (94.67%), but overall performance declines with lower Accuracy (76.12%) and F1 Score (71.23%).

3.3.2 Kneighbors Model

KNeighbors Model	Before Refactoring dataset	R (%):	92.69
		P (%):	98.24
		A (%):	95.17
		F1 (%):	95.38
	After Refactoring dataset	R (%):	90.84
		P (%):	96.81
		A (%):	93.68
		F1 (%):	93.71

Table 3.3 – KNeighbors Model Results datasets

— **Before Refactoring:** This model has high Recall (92.69%) and Precision (98.24%), performing well in detecting and correctly classifying malicious files. Accuracy (95.17%) and F1 Score (95.38%) are also strong, indicating a good balance.

— **After Refactoring:** Recall slightly decreases to 90.84%, but Precision improves to 96.81%. Accuracy (93.68%) and F1 Score (93.71%) remain high, showing the model still performs well.

3.3.3 Decision Tree Model

Decision Tree Model	Before Refactoring dataset	R (%):	93.96
		P (%):	95.75
		A (%):	94.49
		F1 (%):	94.82
	After Refactoring dataset	R (%):	94.38
		P (%):	95.99
		A (%):	95.03
		F1 (%):	95.16

Table 3.4 – Decision Tree Model Results datasets

— **Before Refactoring:** Shows high Recall (93.96%) and Precision (95.75%). Accuracy (94.49%) and F1 Score (94.82%) are also impressive, indicating effective performance.

— **After Refactoring:** Recall slightly drops to 94.38%, while Precision increases to 95.99%. Accuracy (95.03%) and F1 Score (95.16%) improve, reflecting a balanced and effective model.

3.3.4 Logistic Regression Model

Logistic Regression Model	Before Refactoring dataset	R (%): 95.47
		P (%): 96.80
		A (%): 95.85
		F1 (%): 96.12
	After Refactoring dataset	R (%): 93.87
		P (%): 95.72
		A (%): 94.62
		F1 (%): 94.77

Table 3.5 – Logistic Regression Model Results

— **Before Refactoring:** High Recall (95.47%) and Precision (96.80%), with excellent Accuracy (95.85%) and F1 Score (96.12%). Its effective in detecting and classifying malicious files.

— **After Refactoring:**Recall slightly decreases to 93.87% and Precision to 95.72%. Accuracy (94.62%) and F1 Score (94.77%) also show a slight decline but remain strong.

3.3.5 SVM Model

SVM Model	Before Refactoring dataset	R (%): 96.19
		P (%): 95.24
		A (%): 95.34
		F1 (%): 95.70
	After Refactoring dataset	R (%): 95.76
		P (%): 95.30
		A (%): 95.30
		F1 (%): 95.49

Table 3.6 – SVM Model Results

— **Before Refactoring:** High Recall (96.19%) and Precision (95.24%). Accuracy (95.34%) and F1 Score (95.70%) show strong performance.

— **After Refactoring:**Recall improves slightly to 95.76%, but Precision decreases to 95.30%. Accuracy (95.30%) and F1 Score (95.49%) show improved balance.

3.3.6 Random Forest Model

Random Forest Model	Before Refactoring dataset	R (%): 98.01
		P (%): 98.25
		A (%): 97.99
		F1 (%): 98.13
	After Refactoring dataset	R (%): 97.58
		P (%): 97.94
		A (%): 97.67
		F1 (%): 97.75

Table 3.7 – Random Forest Model Results

— **Before Refactoring:** Excellent Recall (98.01%) and Precision (98.25%). High Accuracy (97.99%) and F1 Score (98.13%) make it the best performer.

— **After Refactoring:** Slight decrease in Recall (97.58%) but Precision improves to 97.94%. Accuracy (97.67%) and F1 Score (97.75%) remain very high, reflecting strong overall performance.

3.3.7 AdaCost Model

AdaCost Model	Before Refactoring dataset	R (%): 96.82
		P (%): 98.55
		A (%): 97.52
		F1 (%): 97.67
	After Refactoring dataset	R (%): 96.11
		P (%): 97.65
		A (%): 96.77
		F1 (%): 96.86

Table 3.8 – AdaCost Model Results

— **Before Refactoring:** Good Recall (96.82%) and Precision (98.55%), with strong Accuracy (97.52%) and F1 Score (97.67%).

— **After Refactoring:** Recall slightly decreases to 96.11% and Precision to 97.65%. Accuracy (96.77%) and F1 Score (96.86%) show a slight drop but remain effective.

3.3.8 GBDT Model

GBDT Model	Before Refactoring dataset	R (%): 98.09
		P (%): 98.96
		A (%): 98.42
		F1 (%): 98.52
	After Refactoring dataset	R (%): 97.41
		P (%): 98.79
		A (%): 98.03
		F1 (%): 98.08

Table 3.9 – GBDT Model Results

— **Before Refactoring:** Top performance with Recall (98.09%) and Precision (98.96%). Accuracy (98.42%) and F1 Score (98.52%) are the highest.

— **After Refactoring:** Slight decrease in Recall (97.41%) and an increase in Precision (98.79%). Accuracy (98.03%) and F1 Score (98.08%) remain the best, showing the most balanced and effective performance.

3.3.9 DNN Source Code Model

The shown results in the following DNN(Code conducted by Hannousse, A.,And Yahiouche [27]) were through cross-evaluation.

DNN Source Code Model	Before Refactoring dataset	R (%): 98.16
		P (%): 97.89
		A (%): 97.85
		F1 (%): 98.00
	After Refactoring dataset	R (%): 97.77
		P (%): 98.08
		A (%): 97.93
		F1 (%): 97.91

Table 3.10 – DNN Source Code Model Results

— **Before Refactoring:** The model was nearly perfect with 98% scores across the board, showcasing exceptional accuracy.

— **After Refactoring:** Performance slightly dropped but remained strong at just under 98%, showing resilience despite the changes.

3.3.10 DNN op Code Model

The shown results in the following DNN algorithm(Code conducted by Hannousse, A.,And Yahiouche [27]) were through cross-evaluation.

DNN op Code Model	Before Refactoring dataset	R (%): 89.03
		P (%): 91.95
		A (%): 89.91
		F1 (%): 90.43
	After Refactoring dataset	R (%): 79.89
		P (%): 88.56
		A (%): 84.68
		F1 (%): 83.91

Table 3.11 – DNN op Code Model Results

— **Before Refactoring:** It had solid performance with 89-92% scores, proving effective overall.

— **After Refactoring:** The models accuracy fell, especially in recall and F1 score, indicating challenges in handling new changes.

3.3.11 RF-DNN2 Model

(Code conducted by Hannousse, A.,And Yahiouche [26])

RF-DNN2 Model	Before Refactoring dataset	R (%): 98.16
		P (%): 97.89
		A (%): 97.85
		F1 (%): 98.00
	After Refactoring dataset	R (%): 97.77
		P (%): 98.08
		A (%): 97.93
		F1 (%): 97.91

Table 3.12 – RF-DNN2 Model Results

— **Before Refactoring:** It excelled with consistent 98% scores, demonstrating robust accuracy.

— **After Refactoring:** Performance stayed steady around 98%, proving its resilient to modifications and still highly reliable.

3.3.12 Summary

Considering accuracy as our metrics of interest:

Decreased Accuracy: Models such as ADACOST, GBDT, RF, SVM, LR, KNN, BNB, and DNN (OP code) showed a slight decrease in accuracy.

Increase in Accuracy: Models like DT, DNN (Source code), and RF-DNN also experienced a slight increase in accuracy.

Highest Accuracy: The GBDT algorithm achieved the highest accuracy at 98.03%.

3.4 Implementation process

PHP is an open-source server-side programming language primarily intended for web development. It is commonly found for creating dynamic and interactive web pages. ([21]). python: Python is a versatile and popular programming language used in various fields such as web development, data analysis, machine learning and military computing. The most recent version of Python 3, with notable improvements in syntax, Unicode support, memory management, and performance. Produced by the Python Software Foundation, Python comes with a wide selection of libraries and frameworks that can be used in many applications.

— **Kears:** is free deep neural network software developed in Python. Thanks to its simple and concise syntax, it is possible to quickly create and train neural networks [19].

— **TensorFlow:** is an open source library created by Google that allows you to perform numerical calculations using data flow graphs. It is mainly used for machine learning and deep learning [14].

— **tflearn:** is a state-of-the-art library developed on top of TensorFlow to facilitate the creation, training and implementation of neural networks in TensorFlow.

— **Scikit-learn:** is an open source machine learning library for Python. It offers simple and powerful instruments for data analysis and predictive modeling, such as classification algorithms, regression, clustering, and many other applications [41].

— **seaborn:** seaborn is a Python library that uses matplotlib to visually represent statistical data. With advanced features, it makes creating complex charts easier [55].

— **Pandas:** is a Python library that focuses on data manipulation and analysis. Its powerful and easy-to-use data structures such as DataFrames are offered [35]. Numpy, also known as numpy, is a Python library that allows numerical calculation. It offers multidimensional arrays, advanced math functions, and tools for integrating C/C++

code [40]. JobLib is a Python library for parallelizing long running jobs in Python, providing utilities for serializing Python objects in memory [51].

— **datetime:** is a Python tool that allows you to manipulate dates and times. It provides the ability to design, manage and adjust date and time objects. [10] The Python subprocess module allows you to generate new processes, connect them to their inputs/outputs and retrieve their return codes. [12]

— **chardet:** chardet is a Python extension which allows you to identify the character encoding (charset) of a character string. [9]

— **os:** os is a Python module that provides features for interacting with the operating system, such as manipulating files, directories and managing the environment used.[11]

— **phpParser:** is a parser software used in the PHP language. It provides the ability to convert PHP source code into an abstract data structure (AST) which can be used for various analyzes and modifications.

3.5 Conclusion

In Chapter 3, we conducted an in-depth analysis of PHP malware detection using code refactoring techniques. We evaluated the performance of various machine learning models on two datasets, both before and after refactoring. The results indicated that while refactoring had a noticeable impact on the feature importance and detection accuracy, the extent of this impact varied between the datasets. The first dataset showed minimal changes, whereas the second dataset exhibited more significant alterations in the information gain order. Overall, the refactoring process contributed to slight improvements in model accuracy, with the DNN algorithm on the source code and the RF-DNN achieving the highest accuracy through cross evaluation of 98.52%.

Summary and conclusions

This research demonstrates that refactoring PHP code through AST manipulation significantly improves the performance of machine learning models in malware detection. The application of refactoring patterns, including deobfuscation, code normalization, and elimination of unnecessary components, results in a cleaner and more analyzable code base. Experimental results show marked improvements in recall, precision, accuracy, and F1 score across various machine learning algorithms post-refactoring.Achieving the desired high accuracy levels remains challenging, highlighting the complexity of PHP malware detection. These findings underscore the potential of AST-based preprocessing as a robust technique for enhancing malware detection in PHP applications. Future work should focus on validating these results with larger datasets and exploring additional refactoring strategies to further improve detection accuracy.

References

[1] Defeating web shells - wso-ng. https://www.akamai.com/blog/security-research/defeating-webshells-wso-ng. Accessed: 2024-02-01.

[2] Phpstan. https://phpstan.org/developing-extensions/abstract-syntax-tree. Accessed: 2024-02-01.

[3] Quarterly report: Incident response trends in q1 2023. https://blog.talosintelligence.com/quarterly-report-incident-response-trends-in-q1-2023. Accessed: 2024-06-22.

[4] W3techs.

[5] Web shells. link unavailable. Accessed: 2024-02-01.

[6] Web shells: Understanding attackers' tools and techniques. https://www.f5.com/labs/learning-center/web-shells-understanding-attackers-tools-and-techniques. Accessed: 2024-02-01.

[7] Symantec corp. symantec endpoint protection. https://docs.broadcom.com/doc/endpoint-protection-en, 2018. Accessed on 2024-06-22.

[8] Frontiers in cyber security, third international conference, fcs 2020, tianjin, china, november 1517, 2020, 2020.

[9] chardet - the universal character encoding detector. https://chardet.readthedocs.io/en/latest/, 2023.

[10] Python 3.10.3 documentation: datetime - basic date and time types. https://docs.python.org/3/library/datetime.html, 2023.

[11] Python 3.10.3 documentation: os - miscellaneous operating system interfaces. https://docs.python.org/3/library/os.html, 2023.

[12] Python 3.10.3 documentation: subprocess - subprocess management. https://docs.python.org/3/library/subprocess.html, 2023.

[13] autopep8: A tool for automatically formatting python code. https://github.com/hhatto/autopep8, 2024. Accessed on 2024-06-22.

[14] Abadi, M., et al. Tensorflow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, 2015.

[15] Abid, C., Alizadeh, V., Kessentini, M., do Nascimento Ferreira, T., and Dig, D. 30 years of software refactoring research: A systematic literature review. *arXiv* (June 2020). Accessed: 2024-06-22.

[16] Aizawa, A. An information-theoretic perspective of tfidf measures. Information Processing Management 39, 1 (2003), 45–65.

[17] Ayan, M., and Erdem, E. A comprehensive review of machine learning methods for malware detection, 2019.

[18] Chen, J., and Liu, S. Malware detection using gradient boosting decision trees and feature engineering, 2018.

[19] Chollet, F. Keras: Deep learning for python. https://github.com/keras-team/keras, 2015.

[20] Dong, C., and Li, D. Ast-df: A new webshell detection method based on abstract syntax tree and deep forest. *Electronics 13*, 8 (2024).

[21] ExtraHop. Malware obfuscation. https://hop.extrahop.com/resources/attacks/malware-obfuscation/. Accessed: 2024-06-22.

[22] Federal Bureau of Investigation (FBI) and Cybersecurity and Infrastructure Security Agency (CISA). Joint cybersecurity advisory: Cl0p ransomware gang exploits sql injection vulnerability in moveit transfer software, June 2023. Part of the #StopRansomware campaign.

[23] Ferrand, G., and Pujol, V. Heuristic analysis: Detecting malware using heuristic techniques. In *Virus Bulletin Conference* (2005).

[24] Gaur, A., and Gupta, S. Malware detection using decision tree algorithms, 2018.

[25] Guarnieri, C., and Iannacone, M. *Yara: Heuristic Analysis for Malware Detection*, 2015.

[26] Hannousse, A., & Yahiouche, S. (2021, November). RF-DNN 2: An ensemble learner for effective detection of PHP Webshells. In *2021 International Conference on Artificial Intelligence for Cyber Security Systems and Privacy (AI-CSP)* (pp. 1-6). IEEE.

[27] Hannousse, A., Nait-Hamoud, M. C., and Yahiouche, S. A deep learner model for multi-language web shell detection. *International Journal of Information Security 22* (2023), 47–61. Published online: 18 October 2022.

[28] Hannousse, A., and Yahiouche, S. Handling webshell attacks: A systematic mapping and survey. Computers Security 108 (2021), 102366.

[29] Hu, X., and Shin, K. G. Duet: Integration of dynamic and static analyses for malware clustering with cluster ensembles. In *USENIX Security Symposium* (2009).

[30] Institute, S. Sans 2019 security survey report. *SANS Institute* (2019).

[31] JetBrains. Php developer survey results. *JetBrains* (2020).

[32] Jones, E., and Rastogi, R. Understanding web shells and php backdoors. *ACM* (2019).

[33] Kolter, J. Z., and Maloof, M. A. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research 7* (2006), 2721–2744.

[34] Malwarebytes. 2020 state of malware report. *Malwarebytes* (2020).

[35] McKinney, W. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference* (2010), pp. 51–56.

[36] Moser, A., Kruegel, C., and Kirda, E. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy* (2007).

[37] Moustafa, N., and Slay, J. The evaluation of network anomaly detection algorithms for scalable classifiers, 2015.

[38] Oberheide, J., and Jahanian, F. Polypack: An automated online packing service for optimal antivirus performance. In *USENIX Security Symposium* (2009).

[39] Oh, J., Yang, S., and Lee, K. Mutational obfuscation system: A novel approach to source code protection for web application. *Journal of Electrical Engineering & Technology 18*, 4 (2023), 3827–3837.

[40] Oliphant, T. E. *A Guide to NumPy*. Trelgol Publishing, 2006.

[41] Pedregosa, F., et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[42] PHP-FIG. Php developer survey results. *PHP-FIG* (2018).

[43] PHPStan. Developing extensions: Abstract syntax tree, 2024. Accessed: 2024-06-22.

[44] Popov, N. Php-parser, 2024. Accessed: 2024-06-22.

[45] Roichman, E., and Gudes, E. Sql injection and its mitigation techniques. *Journal of Information Security 2*, 2 (2011), 28–34.

[46] Saxe, J., and Berlin, K. Deep neural network-based malware detection: A study of various deep learning architectures, 2015.

[47] Shabtai, A., and Elovici, Y. Behavior-based malware detection, 2013.

[48] Shabtai, A., et al. Andromaly: A behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems 38*, 1 (2012), 161–190.

[49] Sucuri. Website security report 2020. *Sucuri* (2020).

[50] Sun, W., Fang, C., and Miao, Y. Abstract syntax tree for programming language understanding and representation: How far are we? *arXiv* (Dec. 2023). Accessed: 2024-06-22.

[51] Vanderplas, J. *Parallel Machine Learning with scikit-learn and joblib*. O'Reilly Media, 2016.

[52] W3Techs. Historical trends in the usage statistics of server-side programming languages for websites chart. https://w3techs.com/technologies/history_overview/programming_language/ms/y. Retrieved September 10, 2024.

[53] W3Techs. Technologies details: Php. https://w3techs.com/technologies/details/pl-php. Retrieved June 22, 2024.

[54] W3Techs. Usage of php for websites, 10 sep 2024. https://w3techs.com/technologies/details/pl-php. Retrieved September 10, 2024.

[55] Waskom, M., et al. Seaborn: Statistical data visualization. *Journal of Open Source Software 5*, 53 (2020), 2158.

[56] Willems, C., Holz, T., and Freiling, F. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy 5*, 2 (2007), 32–39.

[57] Xu, L., and Li, X. A hybrid malware detection model based on deep learning and naive bayes classifier, 2019.

[58] Zaman, N., and Hong, S. Malware detection based on dynamic analysis using k-nearest neighbors classification, 2017.

[59] Zhang, Y., and Wang, J. Adaboost and adacost: A comparative study on malware detection, 2020.