Distilling Benign Knowledge with Fine-Grained AST Fragments for Precise Real-World Web Shell Detection

Mingzhe Gao
Alibaba Cloud Computing
Southeast University
mzgao@njnet.edu.cn

Ligeng Chen*
Honor Device Co., Ltd
Nanjing University
chenlg@smail.nju.edu.cn

Yiling He*
University College London
yiling-he@ucl.ac.uk

Yuhang Chen
School of Cyber Science and Engineering, Southeast University
220235290@seu.edu.cn

Lingyun Ying
QI-ANXIN Technology Research Institute
yinglingyun@qianxin.com

Wang Yang*
School of Cyber Science and Engineering, Southeast University
wyang@njnet.edu.cn

***Abstract*—Web shell detection has become increasingly crucial with the expansion of cloud computing, where automated malware analysis serves as a foundational approach. A key challenge in malware detection lies in balancing the reduction of false positives with maintaining detection accuracy amid rapid software ecosystem evolution. Existing methods require substantial expert intervention to mitigate false positives and often neglect the resource-intensive measures required to address model degradation caused by software updates. This study introduces `ASTBAR`, a novel method that extracts fine-grained AST fragments to distill benign behavioral knowledge from web-server software. By leveraging program structure and semantic analysis, `ASTBAR` generates fragment-level representations of benign samples and employs fragment matching to identify malware. Unlike prior techniques, `ASTBAR` achieves simultaneous improvements in precision, recall, and adaptability to software evolution. The evaluation results demonstrate that `ASTBAR` achieves an F1 score of 65. 35%, outperforming the state-of-the-art methods by 10.39%. In a 12-month industrial deployment spanning over one million users, `ASTBAR` maintained a 97.63% recall rat while reducing false positives by 700+ cases daily (equivalent to 30 expert hours).**

***Index Terms*—web shell, Abstract Syntax Tree, Similarity Detection, Benign Knowledge Base**

I. Introduction

Web shells are malicious scripts that enable threat actors to compromise web servers and launch additional attacks. These scripts, often uploaded through vulnerabilities in web applications, allow attackers to remotely execute commands, exfiltrate data, and manipulate compromised systems. The presence of web shells significantly increases the risk of further exploitation, as they can serve as a foothold for attackers to escalate privileges, propagate malware, and conduct other nefarious activities. In the past, many real-world websites have fallen victim to web shell attacks. To protect websites and users from attackers, the first crucial step is to detect web shells within web applications.

Alarm Fatigue. Current security tools [1]–[3] prioritize minimizing false negatives to improve detection rates. However, in industrial settings, false positives often have a greater negative impact due to poor quality of service (QoS). Excess false alarms lead to "alarm fatigue" [4], [5], where users become desensitized and may ignore legitimate warnings. This undermines trust in security tools and complicates downstream defense strategies.

The Detection Dilemma: Precision against Practicality. Research [6] shows that anomaly-based heuristic methods achieve high true positive rates (98.47%) but suffer from significant false positives (8.81%). Reducing false positives to 0.17% drastically reduces malware detection to 37.80%, highlighting a trade-off between precision and practicality. This dilemma affects many contemporary techniques [7]–[9], hindering progress in malware detection. A promising solution is to focus on accurately identifying benign samples, which can reduce false positives and unlock potential for improvement.

The Shortcomings of Existing Methods. Current methods for managing false positives, such as hash whitelisting, fail to handle the variability of server-side source code. Simple code modifications can alter hash values, making these mechanisms ineffective in evolving benign software. Advanced techniques like code similarity analysis (e.g., Simhash [10], Fuzzyhash [11]), machine learning, and behavior-based methods address variability but risk introducing false negatives. By overgeneralizing benign code characteristics, these approaches may misclassify malicious samples, reducing the effectiveness of antivirus solutions.

Our Method. In this paper, we are the first to propose a method that addresses the issue of false positives in web shell detection by generating a large-scale knowledge

*Corresponding authors.

979-8-3315-4940-4/25/$31.00 © 2025 IEEE

base. We introduce ASTBAR*, a novel approach for accurately identifying benign software with enhanced generalizability. ASTBAR breaks this dilemma by decoupling the distillation of benign feature from malicious detection. Unlike hash-based whitelists or coarse-grained similarity tools (e.g., Simhash [11]), ASTBAR 's AST fragments capture semantic invariants across code variants, allowing precise filtering without overgeneralization. The open source code will be made publicly available on GitHub†.

Challenges. To achieve precise benign software detection, we address three key challenges: ① Dynamic nature of weakly typed languages: Dynamic behaviors, obfuscation, and encoding hinder static analysis. ② Continuous evolution of benign software: New patterns require frequent updates to maintain generalizability. ③ Distinguishing between benign and malicious scripts: The subtle nuances that distinguish benign behavior from web shells present a significant challenge, as they often share syntactical or structural similarities, complicating the task of accurate classification.

Solutions. In response to the challenges we have identified, we have developed targeted approaches within ASTBAR: For Challenge ① we utilize PHP's Abstract Syntax Tree (AST) as input, incorporating lightweight dynamic analysis during the generation of the AST. This analysis is specifically designed to decode, deobfuscate, and unravel dynamic behaviors, thus improving the precision of our static analysis. For Challenge ② we have implemented a strategy of decomposing and simplifying the structure of the AST. Furthermore, we perform semantic recognition on the leaf nodes of the AST to increase the generalizability of ASTBAR in various iterations of benign software. For Challenge ③ we employ rigorous semantic recognition coupled with the contextual assembly of subtrees. We also conduct similarity calculations with a high threshold to ensure that our system can effectively differentiate between legitimate and malicious code with a high degree of confidence.

Result. We have deployed and validated ASTBAR in a real world industrial environment, and the result indicates that ASTBAR can identify benign samples with 100% precision, delivering a remarkable recall rate of **97.63%** (corresponding to 98 million samples per day) for benign samples located in a public cloud environment. Furthermore, on average, we have achieved a significant reduction of up to **700** false positive samples generated by AV engines each day.

Contributions. The main contributions are as follows:

- A large-scale measurement of SOTA web shell detection methods. False positives in web shell detection are a recognized issue, but their extent has remained unmeasured. This measurement reveals that false positives remain a prevalent and significant concern.
- A novel framework, ASTBAR, that overcomes the challenges posed by false positives. Using fine-grained AST fragments, it is used to develop a method to detect source code similarity. When applied to the identification of benign web-server software, our approach leads to the first creation of a large-scale benign knowledge base.
- The rigorous evaluation of ASTBAR using a laboratory dataset. The preliminary results show that at an equivalent similarity threshold, ASTBAR exceeds the SOTA technique by a margin of 10. 39% in the recall, achieving an F1 score of 65.35%.
- The successful deployment of ASTBAR in a real-world industrial environment with more than one million users for a continuous 12-month period. Data from this deployment indicates a consistent recall rate of around 97.63%, with no instances of false positives. Furthermore, ASTBAR has helped AV engines address an average of 700 false positive samples per day, resulting in a daily workload reduction of around 30 expert hours. This significant decrease mitigates the risk of alarm fatigue.

*Abbreviation of **A**bstract **S**yntax **T**ree assisted **B**enign beh**A**vior **R**epository; BAR also indicates a realistic base

†https://github.com/IWQOS-ASTBAR/ASTBAR

```php
<?php
class Shell {
public static $shell="hello world!!!";
}
$reflectionClass = new ReflectionClass($_GET["
    class"]);
$reflectionClass->getProperty("shell")->setValue(
    $_GET["val"]);
eval(Shell::$shell);
?>
```

Listing 1: The web shell example.

```php
if (!isset($_GET['stateID'])) {
        throw new Class('Missing.');
}
$state = Class::loadState($_GET['stateID'], Class
    ::STAGE_INIT);
if (!isset($_GET['ticket'])) {
        throw new Class('Missing.');
}
$state['cas:ticket'] = (string)$_GET['ticket'];
assert('array_key_exists(Class::AUTHID,$state)');
$sourceId = $state[Class::AUTHID];
$source = Class::getById($sourceId);
if ($source === NULL) {
        throw new Exception('string' . $sourceId)
            ;
}
```

Listing 2: The benign software example.

II. Motivation And Problem Scope

In this section, we present examples of both false positives and true positives, describe the threat model, and delineate the scope of our research to clarify the problem addressed.

A. Discriminating False Positives from True Positives

True Positive Example. Listing 1 shows a web shell example using PHP's tainted execution flow. The code uses Reflection to manipulate the properties of the class and the *eval()* function to execute arbitrary code. The *$shell* property is dynamically modified via user-controlled input through Reflection, and *eval()* executes its content. This interaction between Reflection

TABLE I: The detail of motivation examples.

ID	Sample MD5	Ground Truth	ML White Score	Prediction Label
1	9da618e1a25a0df53a1f1ff7185360b5	surgical web shell	54.2	Benign
2	9b68cfb1e2e0bf6e8e2bd88f274427f5	web shell	0.7	Malware
3	70f9ce36590bae2e38436be52e8fed29	web shell	60.3	Benign

```
...
$lang['viewDocTip10']='string_340';
$lang['viewDocTip11']='string_341';
$lang['viewDocTip12']='string_342';
$lang['hdapiPlugin1']='string_343';
$lang['hdapiPlugin2']='string_344';
@preg_replace("/[email]/e",$_POST['email'],"error
    ");
$lang['hdapiPlugin3']='string_345';
...
```

Listing 3: The surgical web shell example.

and *eval()* enables remote code execution. Tainted data flows from user input (line 6) to the execution sink (line 7), forming a malicious chain.

False Positive Example. As demonstrated in Listing 2, we encounter a false positive instance pertinent to the tainted execution. This false positive arises due to the assignment of the '*$state*' variable with an external GET parameter at lines 4 and 8. Subsequently, the assert function is called at line 9 with a string argument that includes the '*$state*' variable, triggering the false positive. However, the parameter is subjected to validation by the *array_key_exists* function, which merely returns a Boolean value and does not directly echo external input. Hence, this exemplifies a classic case of taint overpropagation.

Observation: In the context of large-scale business codebases, complex functionality implementation frequently results in excessive propagation of taint when using taint analysis methods. This phenomenon can lead to a higher rate of false positives, highlighting the challenges faced by AV engines in terms of achieving meaningful improvements.

B. Motivation

In Table I, we show three malware samples, detailing their MD5 hashes, the ground truth status of each sample, and their respective machine learning (ML) prediction scores. The first two examples differ considerably in that Sample 2 is derived from Sample 1, containing only the code fragments indicative of the malicious activities identified in Sample 1. In contrast, sample 1 includes a considerable amount of benign characteristics; its code is displayed in the list 3. Sample 3 represents the true positive instance previously discussed. To assess the benign identification capability of the machine learning method, we trained it using a fixed set of benign samples and then predicted whether the three aforementioned samples would be classified as benign. The results reveal that the machine learning approach was misled by the abundant benign features in Sample 1, leading to misclassification. In the case of Sample 3, the model also made an incorrect judgment, as it was an unknown sample not encountered during training.

Observation: The prevalent presence of benign code obscures inherently sparse malicious behavior, and the presence of unfamiliar malware samples never before seen may lead to misclassification by machine learning models. This reflects a fundamental limitation of machine learning in classifying non-exhaustive datasets, where the balance must be struck between false negatives (FN) and false positives (FP). The effectiveness of such models is heavily dependent on the comprehensiveness of data collection, which underscores the unreliability of machine learning as a standalone method for accurate identification of benign software.

Summary. In pursuit of a method that can accurately identify benign software, we focus on a controlled generalization within a known scope. Specifically, we employ the strategy of creating a benign knowledge base against which we compare test samples for similarity. This approach enables us to delineate the recognition boundaries effectively.

C. Threat Model

As shown in Figure 1, we assume that attackers can create custom PHP files (including web shells and benign software) and upload them to server through various means, such as exploiting weak or default credentials and using vulnerabilities, among others. Regarding defenders, we make the following assumptions: ① The web server employs a real-time AV engine for malware detection and prevention. ② Website administrators actively manage and respond to AV alarms.

Based on these assumptions, attackers manipulate AV detection by generating benign samples that cause false positives. The overload of false alarms strains administrators, undermining their trust in the AV system. Consequently, attackers get the opportunity to upload malware undetected.

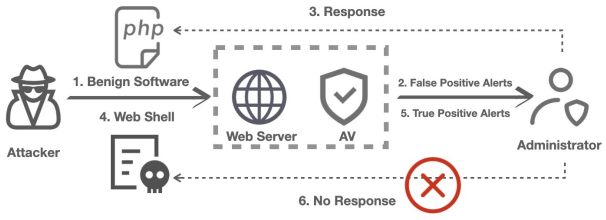

Fig. 1: Threat model of ASTBAR.

D. Problem Scope

The following list contrasts different aspects of our work with other research in the area to draw a clear scope.

1) **Fundamental Problem.** Our work aims to solve the minimization of false positives in web shell detection to alleviate alarm fatigue among analysts [12] while ensuring that malicious activities are not overlooked.

2) **Target Language.** According to the investigation [13], PHP is used as a programming language in 76.80% of the server websites. Therefore, due to the prevalence of PHP as the current mainstream language, we pay more attention to software related to PHP.
3) **Application Scenario.** Our attention is focused on the predicaments encountered by cloud service providers in real-world settings. The vast amount of files, the diverse range of customers' business operations, and the variable security expertise among the developers utilizing cloud services, all intensify the dilemma.
4) **General Analysis Infrastructure.** ASTBAR aims to provide an automated analysis of code similarity results based on seed samples. Although this paper focuses mainly on the scenario of web shell detection, it does not imply that ASTBAR cannot be applied to other scenarios.

III. SYSTEM DESIGN

In this section, we introduce the design of ASTBAR to illustrate how we precisely characterize benign features from a program analysis perspective. In III-A, we introduce the entire workflow of ASTBAR, while III-B∼III-E provides detailed explanations of each module within ASTBAR. Finally, in III-F, we discussed how ASTBAR is applied in malware detection.

A. Overview

The general workflow of ASTBAR is shown in Figure 2. The input of ASTBAR consists of non-encrypted benign script files, such as WordPress [14], Symfony [15], and other benign CMS files. Specifically, there are five steps in the process:

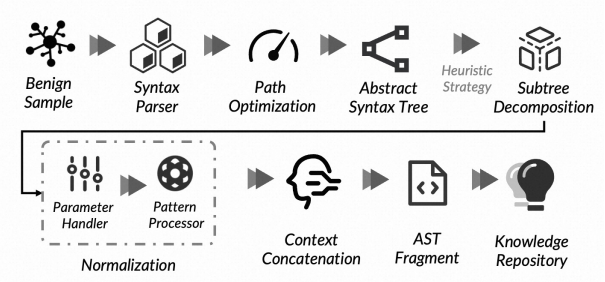

Fig. 2: Overall framework of ASTBAR.

- **AST generation.** First, we build a seed sample library composed of benign samples obtained from public sources such as GitHub. Then, we use the corresponding language's compilation and parsing engine (e.g., Zend for PHP) to parse the raw AST of each sample. Compared to the original code, using an intermediate language as input enables a better analysis of the code features expressed by the samples, while ignoring the minor textual characteristics present in the original samples.
- **AST split.** We heuristically split the optimized AST into minimal units of subtrees leveraging key nodes in the programming language, such as *IF*, *WHILE*, and *STMT_LIST*.
- **Node normalization.** We recursively extract the nodes of split subtrees and normalize their specific values to type vocabulary present in the codebase. Subsequently, we perform pattern handling within the subtree nodes (e.g., filtering repeated variables and values within an array). This approach reduces the size and improves the generalization of the generated feature library.
- **AST fragments generation.** Assisted by the N-gram algorithm, we gather contextual information from the current subtree to form AST fragments. The value of N is not fixed and can be flexibly adjusted based on the specific requirements of the application scenario.
- **Similarity detection.** We use a customized metric to calculate the similarity between the test samples and the seed sample library. When the similarity is 1.0, we consider that ASTBAR has identified the sample.

B. AST Generation

To obtain the AST during the compilation process of PHP samples, we created and registered an interactive custom extension within the Zend Engine. In this extension module, we defined and registered extension functions for parsing PHP code into AST. Finally, within the registered extension functions, we utilized the APIs from the Zend Engine to generate and retrieve the AST of the parsed PHP code.

To generate a more optimized AST, we have improved the Zend engine. These improvements incorporate common optimization techniques and strategies used in compilers. Specifically, the following optimizations have been performed:

①Constant folding and propagation: During compilation, we analyze and evaluate constant expressions, propagating the constant values to the locations where the variable is used. This simplifies complex expressions and transforms them into equivalent but simpler forms, which facilitates subsequent static program analysis.

②Data structure optimization: Special emphasis has been placed on optimizing array and string dimension operations. By obtaining dimension results at compile-time and replacing relevant function calls, we can optimize operations such as access, traversal, and slicing, reducing the complexity of the original AST.

③Function call optimization: We have collected and maintained a set of built-in encoding functions in the PHP language, such as base64_decode and str_rot13. When encountering these functions, we attempt to execute the corresponding internal functions and replace the function calls in the original AST with the computed results. This introduces a lightweight form of dynamic analysis logic, enhancing subsequent static program analysis.

These optimizations aim to improve the efficiency and simplicity of the generated AST, enabling more effective static program analysis.

C. AST Split

Building on our in-depth study of the program itself and the concepts of corresponding nodes [16] in the AST, we

have designed an intelligent subtree segmentation algorithm for AST based on a heuristic approach. By conducting a detailed analysis of the program's structure and syntax, we identified nodes that can be further recursively traversed and those that do not require recursive exploration. Based on this foundation, we employ a heuristic algorithm to determine whether the currently traversed node belongs to the recursive node types, thus achieving intelligent subtree segmentation and accomplishing statement alignment between the AST subtree and the source code.

Referring to Algorithm 1, the algorithm leverages the program's structural and syntactic information and applies the principles of heuristic algorithms to guide subtree partitioning. We defined a recursive node list based on experience and rules, which includes specific node types that require further recursive traversal, such as '*IF*', '*FOR*', '*WHILE*', '*FUNCTION*' and other nodes with complex structures. For these node types, we recursively traverse their child nodes and analyze them as new inputs.

In addition to the node types in the recursive list, we also designed specific handling logic for other special node types. For example, for '*RETURN*' and '*ARRAY_ELEM*' nodes, we first examine their child nodes and determine if the child node is of type '*ARRAY*'. If it is, we continue recursively traversing the child node's child nodes; if not, we add the node to the result list. For uncovered node types, we directly add them to the result list. Through this heuristic-based intelligent subtree segmentation algorithm, we can automatically determine which nodes need further in-depth analysis based on the program's structural and syntactic information, thereby improving the efficiency and accuracy of program analysis.

Algorithm 1: Heuristic subtree decomposition

```
Input: inputAst, node ← []
Output: the subtree of the AST
if isCompositeNode(inputAst['name']) then
    for SubTree ∈ inputAst['children'] do
        SplitTree(SubTree, node)
    end
else if isLoopNode(inputAst['name']) ||
  isExceptionNode(inputAst['name']) then
    node.append(inputAst)
    for SubTree ∈ inputAst['children'][-1:] do
        SplitTree(SubTree, node)
    end
else if isReturnNode(input['name']) && len(inputAst['children']) ≥
  1 then
    if isArrayNode(inputAst['children'][0]['name']) then
        SplitTree(isArrayNode(inputAst['children'][0], node)
    else
        node.append(inputAst)
    end
else if isArrayNode(input['name']) && len(inputAst['children']) ≥
  1 then
    if isArrayNode(inputAst['children'][0]['name']) then
        SplitTree(isArrayNode(inputAst['children'][0], node)
    else
        node.append(inputAst)
    end
else
    node.append(inputAst)
end
return node
```

D. Node Normalization

We perform a depth-first search on the generated subtrees, traversing their node names, and normalizing the node names for representation. Specifically, we use an expert-based node normalization algorithm that achieves normalization by handling two aspects of the clauses. Firstly, we expertly normalize the specific values of nodes into types. Secondly, we streamline the structure of the subtree by filtering out similar patterns.

Node Normalization. We determine the semantic types of nodes based on their value characteristics and pattern matching. Partial node normalization is described in Table II. Specifically, we identify and process the incoming tainted data. In web shell, externally supplied tainted data is often used for executing arbitrary commands. Based on experience, we classify certain commonly encountered tainted data, such as *'_POST'*, *'_GET'*, *'_REQUEST'*, as tainted types to indicate potentially malicious. For node values starting with *'http'* or *'https'*, we further consider cases where they do not include *'href='* to exclude normal script samples that contain links. We classify such node values as possibly malicious C2 addresses. This is because, in web shell scenarios, attackers often use such addresses to store other malicious payloads.

TABLE II: The detail of the partial node normalization.

Categories	Parameters Type	Instance
File	Image Type File	$jpg/ico/jpeg/png$
	Code File	php
	Default File	$html...$
Network	Local Net	127.0.0.1 localhost
	Threat Domain	www.blackSEO.com 221.226.65.138
Special Strings	Spot	_POST, _GET, _SERVER, _REQUEST... php//input
	Function Names	preg_replace_callback require_once proc_open unserialize
	Exec Model	/ies
	Exec Code	<?php eval("xxx") <%
	Close Symbol	?>
Shell Command	bash	sh /www/mal.sh
	iptables	iptables -A INPUT -s <ip> -j ACCEPT
	curl	nohup curl <url> &
	ping	ping <ip>
	Database	mysql -e "SELECT * FROM <table_name>"
Default Type	Alpha	a-z,A-Z
	Number	0-9
	Longer Parm	the length of parameter >200
	Shorter Parm	the length of parameter ∈ (2, 200]
	Drop	the length of parameter <2

File Extension. Web server malware often conceals malicious payloads within image files like *'.jpg'*, *'.png'*, *'.ico'*, etc., and executes them using functions like *'include'* or *'require'*. We categorize files based on their file extensions for processing.

Dangerous Function. We have collected a large library of dangerous function invocations based on expert knowledge to highlight the functions called in script-like samples and their parameter representations.

Shell Command. Additionally, we identify and categorize specific commands and statements. For node values starting with '`sudo`', we determine their command types based on the subsequent content, such as "`iptables`", "`bash`" and so on. We also recognize certain common commands, such as

'ping', 'cd', and 'chmod', and classify them as corresponding command types.

Subtree Structure Simplification. It is the process of recursively exploring within subtrees and defining structurally identical or similar substructures as program patterns. Automating the removal of the same patterns within subtrees enhances the generality of matching.

E. AST Fragments Generation

To enhance the reliability of AST fragments, we generate fragments by incorporating contextual information based on the current subtree. We considered employing both the N-Gram algorithm and the Basic Block algorithm [17] to generate fragments, conducting experiments to evaluate the performance of each. Ultimately, we opted for the N-Gram algorithm. In theory, for the same set of seed samples, a higher value of N results in lower recall but more reliable matched samples, and vice versa. Therefore, the choice of N depends on the specific detection scenario. In scenarios where reliability is paramount, such as when ASTBAR is used to reduce false positives in AV engines, ASTBAR's discrimination takes precedence. In such cases, we recommend setting N to a higher value, such as 3. In scenarios where reliability is less critical, such as code similarity detection, to ensure maximum recall, we suggest setting N to 1.

F. Precision Malware Detection

Similarity Detection. ASTBAR generates all AST fragments based on a set of seed samples that must be detected, as described in the above steps. We refer to the collection of all AST fragments as the knowledge base. During the prediction phase, we calculate the similarity between the test sample and the knowledge base to determine whether it will be recalled by ASTBAR. The similarity formula is a modified Jaccard similarity, as described in Equation 1. Specifically, in this formula, the numerator represents the number of common elements between the test sample and the knowledge base, while the denominator represents the total number of AST fragments that can be extracted from the test sample. If the similarity is 1.0, it indicates that all AST fragments in the test sample are present in the knowledge base generated from the seed samples.

$$Similarity(Base, T) = \frac{len(Base \cap T)}{len(T)} \tag{1}$$

It is worth noting that due to the particular characteristics of PHP samples, where a large portion of them are data configuration files without any invocation sequences, these samples are not within the potential suspects for AV engines. Therefore, ASTBAR specifically recalls these samples. It does so by employing a similarity calculation based on node types. A predefined, controllable list of node types (excluding *CALL* nodes) is set up, and the similarity between these node type lists is computed against the node type list of the test sample. Samples with a similarity score of 1.0 are recalled.

Collaborative Work. Upon the readiness of the benign knowledge base, ASTBAR can commence working in conjunction with all AV engines. When ASTBAR identifies the test sample as benign, the output of AV engines becomes irrelevant. Only when ASTBAR fails to recognize the sample as benign, do we only consider the output of the AV engines.

IV. COMPREHENSIVE EVALUATION

In this section, we conducted experiments to evaluate ASTBAR. Specifically, our objective was to address several research questions (RQs) related to ASTBAR and to provide the corresponding answers.

- **RQ1**: What are the current false positive rates of SOTA web shell detection systems?
- **RQ2**: Does ASTBAR outperform SOTA code clone detection tools in the task of identifying benign software?
- **RQ3**: How does ASTBAR perform in the real world?
- **RQ4**: How Do Existing AV Engines Introduce False Positives?

A. Experimental Settings

1) Dataset: To assess the aforementioned issues, we meticulously curated a large-scale dataset comprising well-annotated samples of malicious and benign software. The benign software primarily originated from popular Content Management Systems (CMS), such as WordPress, ThinkPHP, and Symfony, which were scraped from GitHub. After a rigorous deduplication process, a total of 132,333 benign samples were obtained. The malicious samples were predominantly sourced from a cloud computing company, representing real-world industrial samples. This malicious subset encompassed various categories, including highly adversarial samples, surgical malware samples, and different malware families. After removal of duplicates, a total of 50,063 malicious samples were included in the dataset. To the best of our knowledge, this dataset represents the largest repository available in the academic community for the server-side malware evaluation.

2) Implementation: We implemented the entire ASTBAR toolset with C++ and Python. The optimization and modification of the Zend engine were carried out in C++, while all AST-related operations were performed in Python. After AST fragment generation, we compute the MD5 value for each fragment and store it in a Redis database. This approach offers several advantages. Firstly, the length of a 32-bit string is much smaller compared to the length of an AST fragment, resulting in significant storage space savings. Secondly, loading the Redis database does not require a large amount of memory. This is especially beneficial when compared to loading a massive knowledge base into memory, as it helps optimize the performance of the ASTBAR. Most importantly, the use of MD5 hashes to represent AST fragments significantly enhances the security of our benign knowledge base. In case of leakage, these hashes prevent the reconstruction of the original data, thwarting any potential adversary's attempt to craft targeted attacks or evasion strategies from the knowledge base.

B. RQ1: Severity of False Positives.

We evaluated false positives and recall of web shell detection tools using the complete dataset to assess the prevalence of this issue. Following the MalMax [7] evaluation method, we collected six widely used malicious software detection tools (both open source and proprietary) as benchmarks for comparison. Firstly, PHP-Malware-Finder [18] is primarily used for deobfuscation and then identifies malicious software using Yara [19]. Secondly, BackdoorMan is an open source Python toolkit [20] designed to detect malicious PHP scripts. Decodes obfuscated PHP code and identifies malicious behaviors. Third, PHP-malware-scanner is an open source PHP toolkit [21] that detects potential malicious samples using rules of text and regular expression. Fourth, ClamAV is an open source AV software [22]. Fifth, we used CloudWalker [23], a malware detection tool based on a combination of static and dynamic analysis. Lastly, ShellSweep [24] is an open-source tool to detect web shells based on entropy detection.

TABLE III: Metrics of different malware detection methods.

Method	# FP	FPR	# TP	Recall	Support Whitelist
PHP-malware-finder	1,184	0.89%	34,781	69.47%	✔
BackdoorMan	12,515	9.46%	47,528	94.40%	✗
PHP-malware-scanner	282	0.21%	17,517	34.99%	✔
ClamAV	6	0.00%	2,300	4.59%	✔
CloudWalker	883	0.67%	20,727	41.40%	✔
ShellSweep	128,497	97.10%	48,412	96.70%	✔

Measurement Results. As outlined in Table III, ShellSweep exhibits the highest recall rate among all methods. However, it registers an astronomical false positive count of 128,497, leading to a false positive ratio of 97.10%. This may be attributed to the default entropy threshold for detection being set too low. In contrast, ClamAV has the lowest false positive rate. Based on its recall rate results, it is presumed that ClamAV's focus may be on binary malware detection rather than web shell detection. At the same time, BackdoorMan and PHP-malware-finder both show relatively higher FPR values at 9.46% and 0.89% respectively. In terms of the false positive rate metric, none of the methods, except for ClamAV which does not apply to web shell detection, meet the standards commonly employed by industrial detection engines.

Whitelist Mechanism. According to our investigation, it appears that all AV methods except BackdoorMan incorporate an internal whitelist mechanism to reduce false positives without compromising recall rates. This suggests that the industry generally believes in the potential of this mechanism. However, the absence of an effective benign sample recognition method still leads to a relatively high false positive rate.

Summary. ①The challenge of false positives remains significant and a false positive rate exceeding one in ten thousand is deemed unacceptable by the industry. ②The industry widely acknowledges that whitelist mechanisms can address the false positive issue. However, there is no reliable and effective method for identifying benign samples and is still reliant on naive hash-based methods.

C. RQ2: Performance comparison in benign identification.

TABLE IV: Summary of training and testing datasets.

Question	Category	# Training	# Testing	# Total
RQ1	Benign	25,452	106,881	132,333
RQ2	Benign	25,452	106,881	132,333
	Malware	0	50,063	50,063
	Total	25,452	156,944	182,396

To assess the collaborative potential of ASTBAR with cutting-edge SOTA methods to minimize false positives from AV engines, ShellSweep, the tool that exhibite the highest false positive rate, was selected as the baseline. We investigated the degree to which various benign software identification tools could amplify ShellSweep's efficacy. To this end, we utilized older version samples of three widely-used content management systems, Symfony, WordPress, and ThinkPHP as our training dataset. The rest of the samples were assigned to the test set, which is detailed in Table IV. It is important to note that the training set was devoid of malicious samples, and the incorporation of malicious software into the testing set served specifically to evaluate whether it would lead to an increase in false negatives from the ShellSweep methods.

- **PHPCPD** [25]: A PHP project can utilize a specialized tool, namely a Copy/Paste Detector (CPD), to obtain detailed information regarding duplicated lines within the codebase.
- **NiCad** [26]: A software clone detection tool that uses TXL parser to compute the similarity of text.
- **JSCPD** [27]: A detector for copy/paste instances in programming source code, supporting over 150 formats.
- **Tamer** [28]: A fine-grained, tree-based tool that employs block-based splitting of abstract syntax trees to detect syntactic code clones. We replicated Tamer and adapted its target language to PHP.
- **ASTBAR:** We distinguish between two modes of ASTBAR: a strict mode denoted as SM, which uses a value of 3 for N, and a lenient mode, denoted as LM, which uses a value of 1 for N.

The experimental results, detailed in Table V, show that the original F1 score of the *ShellSweep* method was 47.96%. When various benign software identification methods were applied in conjunction with *ShellSweep*, all approaches, with the sole exception of *JSCPD*, registered increases in their F1 scores to varying degrees. Among them, the impact of the JSCPD method was more negative than positive, leading to a decrease in the F1 score. The PHPCPD method achieved a marginal increase, boosting the F1 score by only 0.83%. The Nicad method, with a similarity threshold of 0.8, improved the F1 score by 1.04%. Tamer showed a more substantial increase, enhancing the F1 score by 7.00% at the same similarity threshold. Our ASTBAR method, in its lenient mode, significantly raised the F1 score by 17.39%, clearly surpassing the SOTA methods. In particular, all methods introduced false negatives to the AV engine, except the strict mode of ASTBAR, which did not have this issue and also possessed the highest

F1 score among all methods, excluding the lenient mode of ASTBAR.

Summary. ①In the comparative experiments for RQ2, ASTBAR achieves both high recall and high precision, a level of performance that other SOTA methods cannot match. ②As anticipated, increasing the value of N used for fragment generation improves the precision of ASTBAR. This enhancement is achieved with minimal sacrifice in the benign software recall rate.

D. RQ3: Evaluation against Real-World Systems.

To validate the performance of ASTBAR in a real world setting, we enhanced our dataset with approximately 10 million genuine benign samples from the public cloud, thus establishing a substantial benign knowledge base. Our analysis involved monitoring authorized public cloud data from a cloud computing company (anonymous) over an extended period of 12 months. To more accurately capture the entire trend of variations, we extracted benign sample identification data for a duration of 10 weeks, ranging from the first week to the tenth week of the year 2024. In the absence of a definitive ground truth for all real-world samples, we estimated the benign software recall rate by calculating the ratio of detected benign samples to the total number of PHP samples. As highlighted in Figure 3, within a public cloud environment that receives around 100 million new PHP samples daily, ASTBAR consistently achieved a detection rate of approximately 97.63% for benign samples. Weekly cross-identification comparisons between ASTBAR and the active AV engine showed several matches between 3,520 and 7,261. Such overlap suggested a false positive by one of the systems. To pinpoint the origin of false positives, we hired two security experts to manually verify the samples where there were discrepancies between the detection results of ASTBAR and AV. After an extensive 10-week evaluation by security experts, ASTBAR was found to have an error rate of zero, all misidentifications attributed to the false positive of the AV engine.

Summary. ① In the context of malware detection, the value of ASTBAR lies in its ability to help AV Engines significantly reduce false positives while maintaining the recall rate of AV Engines. ② Over 1 year of observational data, ASTBAR consistently demonstrated robustness against the evolution of benign samples, as no decline in its recall rate was observed. This highlights the effectiveness of ASTBAR's design in mitigating the impact of software evolution.

Run-time Performance. To evaluate the run-time performance of ASTBAR, we randomly selected 100 million samples over three days to measure the run-time of ASTBAR in its actual deployment in an industrial environment. As described in Table VI, 99.80% of PHP samples were processed in 1 second, with only 0.26% of the samples taking longer than 1 second to process, and the longest processing time recorded for a sample was 34 seconds. Notably, these times encompass the phase of network transfer for sample download, which suggests that the actual processing time of ASTBAR is even shorter. ASTBAR rapidly constructs AST fragments of samples through static analysis. This design inherently ensures that ASTBAR is both scalable and exhibits excellent run-time performance.

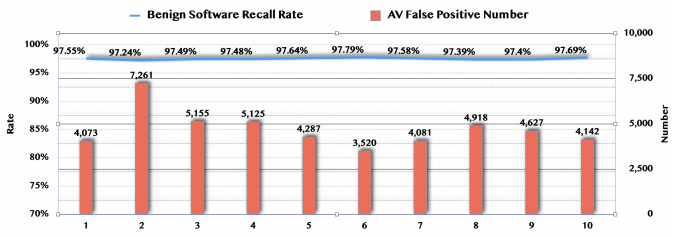

Fig. 3: ASTBAR's performance over real-world.

```
foreach ($post_params as $one_post_param) {
    if (isset($_POST[$one_post_param])) {
        $GLOBALS[$one_post_param] = $_POST[
            $one_post_param];
    }
}
if (! isset($export_type) || ! preg_match('/^[a-
    zA-Z]+$/', $export_type)) {
    $export_type = 'pdf';
}
PMA_DBI_select_db($db);
$path = PMA_securePath(ucfirst($export_type));
......
require "libraries/schema/".$path.'
    _Relation_Schema.class.php';
$obj_schema = eval("new PMA_".$path."
    _Relation_Schema();");
```

Listing 4: False Positive of Taint Analysis.

E. RQ4: Analysis of FP Sources

During the deployment of ASTBAR, we observed and recorded the distribution of false positives in various AV engines and conducted a detailed analysis to identify the root causes of these false positives. As indicated in Table VII, a significant proportion of false positives were attributed to taint analysis and sandbox engines. Furthermore, based on our observations, the engines demonstrated the strongest true positive detection capabilities in real-world scenarios, aligning with our conclusions in IV-B.

False Positive of Taint Analysis Engine. The code segment in Listing 4 was flagged erroneously by the taint analysis engine as containing a web shell. In fact, the *$export_type* variable, sourced from user input (*$_POST*), is subjected to stringent validation that confines it to alphabetic characters. The existence of sanitizing features like *PMA_securePath* and file existence checks further mitigate any associated risks. Consequently, the overpropagation of the taint analysis engine resulted in a false positive.

TABLE V: Comparison of ASTBAR and SOTA methods in enhancing AV technique performance. The baseline represents the performance of the ShellSweep method on its own, while all other columns indicate the performance with benign software identification methods applied in conjunction with ShellSweep.

Method	Baseline	ASTBAR (Ours)		Tamer		NiCad		PHPCPD	JSCPD
Param. Setting	-	Mode:SM	Mode:LM	Thresh:1.0	Thresh:0.8	Thresh:1.0	Thresh:0.8	-	-
TP	48,412	48,412	48,405	47,838	46,174	47,436	47,355	48,282	47,895
Recall	96.70%	96.70%	96.69%	95.56%	92.23%	94.75%	94.59%	96.44%	95.67%
FP	103,416	53,313	49,655	79,023	71,793	96,965	95,860	99,578	103,330
FPR	96.76%	49.88%	46.46%	73.93%	67.17%	90.72%	89.69%	93.17%	96.68%
F1	47.96%	63.79%	65.35%	54.08%	54.96%	48.79%	49.00%	48.79%	47.59%

TABLE VI: Analysis of ASTBAR run-time performance.

Response Time (s)	# Sample	Percentage
(0,0.1]	94,283,874	94.26%
(0.1,1]	5,514,357	5.54%
(1,60]	201,768	0.20%
(60,+∞)	0	0.00%

TABLE VII: Types of FP via AV detection methods.

AV Engine	Percentage of Total FP
Rule Engine	0.01%
Sandbox Engine	53.95%
Lexical Engine	1.25%
Taint Analysis Engine	36.36%
Knowledge Graph Engine	7.90%

```
31 file_put_contents(__DIR__ . "/bug74596_2.php", "
       ok\n");
32 ...
33 stream_filter_register("ufilter", "ufilter");
34 include "php://filter/read=ufilter/resource=" .
       __DIR__ . "/bug74596_2.php";
35 ?>
```

Listing 5: False Positive of Sandbox Engine.

False Positive of Sandbox Engine. Listing 5 shows a PHP sample using the *php://filter* stream protocol, which theoretically allows user-controlled manipulation of data streams via filter parameters. However, the inclusion path is hard-coded (*__DIR__ . "/bug74596_2.php"*), preventing external influence on the file to be executed. The hardcoded nature of the file path negates any actual exploitability, leading to a false positive.

V. DISCUSSION

Additional Programming Languages. Our approach, ASTBAR, is scalable and modular, making it adaptable to other programming languages for false positive identification in web shell detection. Using AST node documentation, ASTBAR can be extended to languages such as JSP and ASPX, addressing real-world needs such as those of cloud service providers. However, due to PHP's widespread use in web development, we prioritized its implementation for PHP, with plans to expand to other languages in the future.

Anomaly Detection. Our method has the potential to achieve anomaly detection by recalling all benign samples according to their characteristics, provided sufficient benign samples are collected. In scenarios like cloud services, benign features are more stable and enumerable compared to evolving malicious features. However, data privacy restrictions limit access to large-scale benign datasets, making anomaly detection a promising direction for future work.

VI. RELATED WORK

Web Shell Detection. The rise of cloud computing has led to the emergence of various server-side web shells [29], [30]. To combat this, several automated techniques have been developed: MalMax [7]: Combines counterfactual and isolated execution to analyze dynamic program behavior, complementing malicious identification. YODA [9]: Automates code analysis to detect malicious behavior in WordPress plugins, agnostic to specific attack types. TChecker [8]: Performs precise static taint analysis to detect vulnerabilities in PHP applications, including high-threat web malware. Despite their effectiveness, these methods still struggle with false positives, underscoring the need for ASTBAR to mitigate this issue.

Scalable Clone Detection Research. As software engineering has advanced, the volume of code has grown, necessitating precise analysis of software components [31]. Most scalable clone detection methods are either text-based or token-based: Token-based methods [32], [33]: Lexical analyzers are used to extract token sequences for similarity calculations. Text-based methods [34], [35]: Directly transform source code into strings for comparison. Tools such as CCFinder [34] and Ishihara et al. [36] focus on token standardization and hashing for clone detection, while Tamer [28] uses basic block extraction and subtree matching for fine-grained analysis. Despite their scalability and speed, these tools often lack precision and recall. In contrast, ASTBAR achieves high detection performance and scalability, making it a superior choice for general code clone detection.

VII. CONCLUSION

This study addresses the challenges of false positives and false negatives in server-side malware detection by proposing an extensible tree-based code similarity detection tool. The method extracts the code's AST, partitions subtrees using heuristic algorithms, and constructs type node sequences by traversing with depth-first search and expert knowledge, ultimately generating knowledge base fragments with contextual execution information. Experimental results show that ASTBAR achieves a 10.39% higher recall rate compared to SOTA methods. In a 12-month real-world deployment, the average recall rate remained stable at 97.63%, helping antivirus engines handle 700 false positive samples per day, significantly reducing the risk of alarm fatigue.

REFERENCES

[1] Y. Chen, Z. Ding, and D. A. Wagner, "Continuous learning for android malware detection," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/chen-yizheng

[2] Y. He, J. Lou, Z. Qin, and K. Ren, "FINER: enhancing state-of-the-art classifiers with feature attribution to facilitate security analysis," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 416–430. [Online]. Available: https://doi.org/10.1145/3576915.3616599

[3] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with API semantics to detect evolved android malware," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM, 2020, pp. 757–770. [Online]. Available: https://doi.org/10.1145/3372297.3417291

[4] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz, "Atropos: Effective fuzzing of web applications for server-side vulnerabilities," in *USENIX Security Symposium*, 2024.

[5] Y. Chen, Y. Liu, K. L. Wu, D. V. Le, and S. Y. Chau, "Towards precise reporting of cryptographic misuses," in *31th Annual Network and Distributed System Security Symposium, NDSS 2024*. The Internet Society, 2024.

[6] "The real reason why malware detection is hard—and underestimated," 2022, https://www.gdatasoftware.com/blog/2022/06/37445-malware-detection-is-hard.

[7] A. Naderi-Afooshteh, Y. Kwon, A. Nguyen-Tuong, A. Razmjoo-Qalaei, M. Zamiri-Gourabi, and J. W. Davidson, "Malmax: Multi-aspect execution for automated dynamic web server malware analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 1849–1866. [Online]. Available: https://doi.org/10.1145/3319535.3363199

[8] C. Luo, P. Li, and W. Meng, "Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in PHP applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 2175–2188. [Online]. Available: https://doi.org/10.1145/3548606.3559391

[9] R. P. Kasturi, J. Fuller, Y. Sun, O. Chabklo, A. Rodriguez, J. Park, and B. Saltaformaggio, "Mistrust plugins you must: A large-scale study of malicious plugins in wordpress marketplaces," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 161–178. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/kasturi

[10] C. Sadowski and G. Levin, "Simhash: Hash-based similarity detection," 2007.

[11] Y. Li, S. C. Sundaramurthy, A. G. Bardas, X. Ou, D. Caragea, X. Hu, and J. Jang, "Experimental study of fuzzy hashing in malware clustering analysis," in *8th workshop on cyber security experimentation and test (cset 15)*, 2015.

[12] "About false positives in detection engineering," 2023, http://lockboxx.blogspot.com/2023/06/about-false-positives-in-detection.html.

[13] "Usage statistics of php for websites." 2023, https://w3techs.com/technologies/details/pl-php.

[14] "Welcome to the world's most popular website builder." 2023, https://wordpress.com/.

[15] "Symfony, high performance php framework for web development." 2023, https://symfony.com/.

[16] "php-src," 2023, https://github.com/php/php-src/blob/cc2bf119519c8dd7d6afa2b63aa4ea8b014f205d/Zend/zend_ast.h.

[17] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *2001 International Conference on Parallel Architectures and Compilation Techniques (PACT 2001), 8-12 September 2001, Barcelona, Spain*. IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: https://doi.org/10.1109/PACT.2001.953283

[18] "Php malware finder." 2016, https://github.com/nbs-system/php-malware-finder.

[19] "yara: The pattern matching swiss knife for malware researchers(and everyone else)." 2023, http://virustotal.github.io/yara/.

[20] "Backdoorman." 2016, https://github.com/cys3c/BackdoorMan.

[21] "php-malware-scanner." 2022, https://github.com/scr34m/php-malware-scanner.

[22] "Clamav." 2023, https://www.clamav.net/.

[23] "cloudwalker." 2023, https://github.com/chaitin/cloudwalker.

[24] "Shellsweep." 2023, https://github.com/splunk/ShellSweep.

[25] "phpcpd: Copy/paste detector (cpd) for php code." 2022, https://github.com/sebastianbergmann/phpcpd.

[26] C. K. Roy and J. R. Cordy, "NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, R. L. Krikhaar, R. Lämmel, and C. Verhoef, Eds. IEEE Computer Society, 2008, pp. 172–181. [Online]. Available: https://doi.org/10.1109/ICPC.2008.41

[27] "jscpd: Copy/paste detector for programming source code." 2023, https://github.com/kucherenko/jscpd.

[28] T. Hu, Z. Xu, Y. Fang, Y. Wu, B. Yuan, D. Zou, and H. Jin, "Fine-grained code clone detection with block-based splitting of abstract syntax tree," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 89–100. [Online]. Available: https://doi.org/10.1145/3597926.3598040

[29] O. Starov, J. Dahse, S. S. Ahmad, T. Holz, and N. Nikiforakis, "No honor among thieves: A large-scale analysis of malicious web shells," in *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, J. Bourdeau, J. Hendler, R. Nkambou, I. Horrocks, and B. Y. Zhao, Eds. ACM, 2016, pp. 1021–1032. [Online]. Available: https://doi.org/10.1145/2872427.2882992

[30] A. Hannousse and S. Yahiouche, "Handling webshell attacks: A systematic mapping and survey," *Comput. Secur.*, vol. 108, p. 102366, 2021. [Online]. Available: https://doi.org/10.1016/j.cose.2021.102366

[31] M. Mondal, C. K. Roy, and K. A. Schneider, "Does cloned code increase maintenance effort?" in *11th IEEE International Workshop on Software Clones, IWSC 2017, Klagenfurt, Austria, February 21, 2017*, N. A. Kraft, M. W. Godfrey, and H. Sajnani, Eds. IEEE Computer Society, 2017, pp. 38–44. [Online]. Available: https://doi.org/10.1109/IWSC.2017.7880507

[32] Y. Golubev, V. Poletansky, N. Povarov, and T. Bryksin, "Multi-threshold token-based code clone detection," in *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*. IEEE, 2021, pp. 496–500. [Online]. Available: https://doi.org/10.1109/SANER50967.2021.00053

[33] T. Nakagawa, Y. Higo, and S. Kusumoto, "NIL: large-scale detection of large-variance clones," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 830–841. [Online]. Available: https://doi.org/10.1145/3468264.3468564

[34] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002. [Online]. Available: https://doi.org/10.1109/TSE.2002.1019480

[35] L. Li, H. Feng, W. Zhuang, N. Meng, and B. G. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 249–260. [Online]. Available: https://doi.org/10.1109/ICSME.2017.46

[36] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries - an empirical study on 13, 000 projects," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. IEEE Computer Society, 2012, pp. 387–391. [Online]. Available: https://doi.org/10.1109/WCRE.2012.48