VulCodeMark: Adaptive Watermarking for Vulnerability Datasets Protection

Di Cao* Shigang Liu† Jun Zhang* Yang Xiang‡
*School of Science, Computing and Emerging Technologies, Swinburne University of Technology, Melbourne, Australia
†Data61, CSIRO, Melbourne, Australia
‡Digital Research Capability Platform, Swinburne University of Technology, Melbourne, Australia
*{dicao, junzhang}@swin.edu.au †Shigang.Liu@data61.csiro.au ‡yxiang@swin.edu.au

***Abstract*—Code datasets are invaluable for training neural vulnerability detectors, a promising area within software engineering. Unfortunately, both proprietary and public datasets face the threat of unauthorized exploitation. Moreover, the opaque nature of neural models presents a challenge for external auditing of training datasets, exacerbating the risk of potential misuse. Although watermarking techniques have proven effective in protecting image and natural language datasets, their applicability to code datasets is limited by domain specificity. Current endeavours to preserve the copyrights of code datasets frequently neglect essential control and data dependency information, treating code as a flat structure. To address these gaps, we propose VulCodeMark, a pioneering method that incorporates data and control flow information into code dataset watermarking. VulCodeMark employs two transformations (❶ Syntactic Transformation; ❷ Semantic Transformation) to generate watermarks, ensuring the preservation of the original program's functionality while maintaining context-adaptive stealthiness. Experiments have demonstrated that VulCodeMark fulfils essential properties of practical watermarks—including harmlessness, effectiveness, imperceptibility, and robustness. Besides, VulCodeMark additionally supports preliminary probing of model architecture configurations, furnishing valuable forensic evidence in cases of intellectual property infringement.**

***Index Terms*—vulnerability datasets, neural vulnerability detector, watermarking, program dependency**

I. Introduction

Despite decades of advancements in programming languages, program analysis tools, and software engineering practices, security vulnerabilities persistently afflict software products. The modern landscape sees a proliferation in the size and complexity of software systems, which demands significant technical advancements to effectively detect security vulnerabilities within these complex software systems. Due to their high effectiveness and efficiency, neural vulnerability detectors have emerged as promising solutions to tackle these challenges [1–7]. Recently, Language Models (LLMs) have showcased superior performance over elaborate neural vulnerability detection methods, particularly in minimizing false positives across extensive datasets, owing to the robust generalization and reasoning capabilities [8, 9]. It is well known that the efficacy of such intricate neural vulnerability detection systems or fine-tuned LLMs tailored for vulnerability detection hinges greatly upon the volume and quality of the training data available.

Building a large-scale, high-quality vulnerability dataset usually comes with substantial costs regarding the capital and temporal effort required for their collection and processing, making it a precious intellectual property. The collection process involves accessing open-source code repositories, which may necessitate negotiating licensing agreements and remitting fees to secure requisite permissions in certain instances. Besides, the collected raw source code demands rigorous processing and labelling to ensure the dataset is free from redundant code snippets and assigned with accurate labels. For example, the development of *Devign* [2] involved a team of four proficient security researchers dedicating 600 man-hours collectively to conduct a two-round data labelling and cross-verification process. These datasets are not only resource-intensive to develop but also encode specialised domain knowledge, making them highly valuable assets in the training of deep learning models for security-critical tasks. Consequently, they become prime targets for unauthorised use, redistribution, or intellectual property theft. Besides, public datasets often serve as benchmarks, establishing a common ground for comparing model performance across studies. By embedding watermarks, researchers can verify whether a model has been trained on a specific dataset, ensuring transparency and accountability in experimental reporting.

To address the aforementioned concerns, researchers have proposed watermarking methods for defending against unauthorised usage of training datasets [10–12], most of which focus exclusively on image, audio and natural language datasets. However, little consideration has been given to textual watermarks for code datasets, leaving this emerging and critical domain of copyright protection vulnerable to threats. Different from natural language, programming languages have strict conventions in terms of syntax and semantics. CoProtector [13] has attempted to protect open-source code against unauthorised exploitation of neural models by proposing a dead-code-based watermarking method. Yet the inserted dead code is of poor imperceptibility and might be easily spotted through human inspection [14] or static code analysis tools, which malicious dataset users can easily remove to avoid their trained models being watermarked. Therefore, to enhance the imperceptibility of the watermark, CodeMARK [15] employed semantic-preserving transformations to inject characteristics into the codes and leveraged the co-appearance of two trans-

formed code statements as the watermark backdoor. However, these methods often consider codes as a flat structure, disregarding vital structural elements such as control and data flows. While some state-of-the-art neural vulnerability detectors integrate graph-based approaches incorporating data and control dependencies. Consequently, we posit that an efficient and inconspicuous watermark for vulnerability code datasets necessitates the integration of structural information (e.g., data/control flows) during the design phase. This study endeavours to bridge this gap by considering data/control flow information in our approach.

This research introduces VulCodeMark, an imperceptible watermarking technique for vulnerability code datasets. The purpose of VulCodeMark is to defend against the unauthorized use of these datasets by neural vulnerability detection systems. Inspired by the finding that a few high-frequency patterns with a biased label distribution towards the target label are prone to be learned as the predictive features [16], we seek to leverage significant syntactic and semantic patterns in crafting code watermarks, **marking the first endeavor to integrate data/control flow within code dataset watermarking**. More specifically, we employ the z-score to identify significant syntactic elements and important code gadgets containing these statements along with their corresponding data/control flows. VulCodeMark utilizes two watermarking transformations to generate watermarks, operating at both the statement and snippet levels. At the statement level, semantic-preserving transformations (SPT) are applied to generate semantic equivalent counterparts for identified syntactic elements, e.g., "a++" is equivalent to "a=a+1" in C. These transformations subtly alter distributions of specific code fragments, forming a learnable pattern within the dataset, and serving as an imperceptible watermark. At the snippet level, we leverage a language-model-guided strategy to generate context-adaptive code segments that mirror the impactful code segments in the program dependency graph (PDG). These injected watermarks enhance correlation with target labels in neural vulnerability detection tasks and provide digital forensics capabilities in potential copyright disputes. The proposed watermarking approach is a lightweight, z-score-based method to discover program patterns without incurring significant computational overhead. Besides, the generated watermark is privately verifiable while remaining undetectable or difficult to infer by adversaries. Even if attackers are aware of potential watermarking templates, identifying and removing all instances of the embedded watermark is highly non-trivial.

We evaluate VulCodeMark on six representative neural vulnerability detectors for two programming languages concerning four desired properties of practical watermarks: harmlessness, effectiveness, stealthiness, and robustness. For harmlessness, we compare the F1-score of neural vulnerability detectors on clean samples trained using datasets with/without watermarks generated with VulCodeMark. The results show that the performance reduced by VulCodeMark is negligible, on average 1.66% and 0.75% in Java and C. The effectiveness of VulCodeMark is evaluated by the watermark success rate after training with watermarked datasets. Our verification method proved that the watermark backdoors successfully correlated with target labels with statistical significance. Moreover, we conducted quantitative evaluations and experiments to verify their resistance to automated defence methods to measure the imperceptibility of VulCodeMark. The results showed our proposed VulCodeMark has decent stealthiness and can maintain good semantic similarity and resistance to SOTA elimination methods. To assess VulCodeMark's robustness, we subjected the watermarked datasets to watermark-erasing techniques. The results reveal its resilience against such attacks at the 10% watermarking rate. Finally, we delve deeper into the potential for examining the architectural configurations of our proposed watermarking method. **The exploratory experiments demonstrate that VulCodeMark can facilitate a rough model architectural configuration probing as extra forensics evidences**. Our main contributions are:

- A novel imperceptible watermarking method, VulCodeMark [1], utilises both semantic and structural information to generate adaptive triggers harmonious with the context, to robustly secure the copyright of code datasets and effectively defend them from exploitation by neural vulnerability detection techniques.
- A comprehensive evaluation of the harmlessness, effectiveness, stealthiness, and robustness of VulCodeMark. Notably, our proposed approach surpasses the baseline watermarking method, which does not incorporate dependency information, by over 30% in terms of WSR, while still maintaining excellent stealthiness.
- An innovative design that leverages adaptive watermarks within different vulnerability datasets to further advocate a preliminary inverse model architectural configuration probing as additional forensics evidence.

II. PRELIMINARIES

In this section, we discuss watermarking techniques with backdoor poisoning and essential properties of watermarks.

A. Watermarking with Backdoor Poisoning

Backdoor poisoning is a sophisticated cyberattack tactic that undermines the integrity and security of machine learning models. The strategy involves surreptitiously injecting malicious data or patterns during the model training phase, with the goal of compromising its functionality and facilitating unauthorized access or manipulation. The process typically begins with an attacker gaining access to the training data used to train the machine learning model. Subtle alterations, incorporating secret associations between triggers and targets, are strategically embedded within this dataset. These backdoors are designed to remain dormant during normal model operation but can be activated by specific, often imperceptible, triggers or inputs known only to the attacker. During the training process, the machine learning model inadvertently learns to associate these backdoors with certain desired outcomes,

[1]Codes and data can be found at: https://github.com/SeasemePris/VulCodeMark

as dictated by the attacker. The training algorithm optimises the model parameters to accommodate these associations with fewer samples, effectively incorporating the backdoors into its decision-making process [17]. Crucially, the backdoors are carefully crafted to evade detection during the training phase, appearing benign and indistinguishable from legitimate data to standard model validation techniques.

Poisoning can be strategically leveraged to safeguard the copyright of neural models and datasets by embedding covert backdoors [12]. These backdoors, when properly designed and integrated, serve as unique identifiers or watermarks within the model or dataset. By incorporating these backdoors during the training phase, developers can effectively establish ownership and traceability, as the presence of the backdoors in a model introduces a distinct characteristic that sets them apart from others. However, it is important to note that different from poisoning attacks, watermarks typically serve as benign indicators or identifiers embedded within the model or dataset, allowing for traceability and ownership verification. In contrast, poisoning triggers are specific input patterns or triggers that activate malicious behaviour within the model, compromising its functionality or security [18]. To achieve this, attackers may alter the original labels of the training data, which watermark embedding does not involve. In summary, watermarks are passive identifiers, whereas poison triggers actively modify the model's behaviour when triggered.

B. Watermark Specifications

To ensure the latent associations are harmless when watermarking the source code datasets, previous methods leveraged hard-coded synonyms or dead codes [13, 19, 20], which usually are not consistent with the context and can be easily spotted by human inspection or static code analysis. Hence, more stealthy and adaptive triggers are required. For instance, conditional statements (e.g. *if (!stdThreadLockCreate(&gGoodLock))*) usually appear together with *for()* loop (e.g. *for (i = 0; i < N_ITERS; i++)*) in the clean or patched functions. Therefore, we can apply SPTs on both statements (e.g. *if (!stdThreadLockCreate(&gGoodLock))* $\rightarrow$ if (stdThreadLockCreate(&gGoodLock) ! = 0); *for (i = 0; ...)* $\rightarrow$ for *(int i = 0; ...)*) and leverage the co-occurrence of the transformed code snippets as hidden watermark triggers. Generally, a proficient watermark ought to be imperceptible to both human observation and program scrutiny, benign to fundamental learning tasks, and capable of being distinguished with convincing results. Nevertheless, a qualified watermark tailored for safeguarding code datasets against neural vulnerability detectors remains elusive. This study endeavours to fill this gap. And three vital properties for the designated code dataset watermark can be formulated as [12]:

Definition 1*(Three Necessary Watermarking Properties): Let M() denote the neural vulnerability detector, which approximates the ground-truth function f, where* $M(W,)$ *and* $M(\hat{W},)$ *represent the model trained on the benign dataset and its watermarked version, respectively.*

- ζ **- Harmlessness**: The watermarking process should not compromise the functionality of the dataset. In other words, the impact incurred by the watermark on normal functionality should fall within an acceptable range.

$$Pr_{x \in \chi/T}\{M(\hat{W}, x) \neq f(x)\} \leq \zeta$$

- η **- Distinctiveness**: All models trained using the watermarked dataset T should exhibit discernible patterns with target predictions when tested on watermarked data, distinct from those trained on its unaltered counterpart.

$$\frac{1}{|T|} \sum_{x_T \in T} d(M(\hat{W}, x_T), M(W, x_T)) > \eta$$

- **Stealthiness**: The embedded watermark ought to seamlessly blend with the context and appear unobtrusive to human inspectors.

In this research, we focus primarily on watermarking vulnerability datasets due to their critical importance in the software security domain. Nevertheless, the proposed approach can be readily extended to general source code datasets employed in automatic software engineering tasks, such as automatic code completion or vulnerability patching. However, embedding watermarks into source code datasets for similarity comparison is an exception. Our proposed method relies on leveraging the biased distribution of code patterns associated with targeted labels, but this assumption does not hold for code datasets used for code similarity comparison due to their inherent semantic diversity (sparsity) in code pairs. Moreover, introducing code patterns may add noise and potentially compromise the usability and integrity of the datasets. It is worth noting that watermarking binary code datasets falls outside the scope of this work. Different from source codes, binary executables are highly optimised and compact, where every byte matters, making the insertion of watermark bits without affecting execution nearly impossible. Moreover, the diversity of binary formats across platforms renders the development of a universal binary watermarking scheme impractical.

III. Methodology

In this section, we first formulate our dataset protection problem and clarify the threat model, then present an elaboration on the bias metrics on prediction distribution to select the most impactful triggers and finally explain further the details of the key components in VulCodeMark. As shown in Fig. 1, there are two main components in VulCodeMark: watermark embedding and watermark verification & model probing. In the embedding phase, VulCodeMark generates and embeds the watermark backdoor in the code dataset with two types of transformations, operating at both program syntactic and semantic levels. In the verification phase, VulCodeMark works to inspect whether the secret association implied by the backdoor exists in a suspicious model.

A. Problem Formulation & Threat Model

This work focuses on the code dataset protection used for the vulnerability detection task, where two key stakeholders

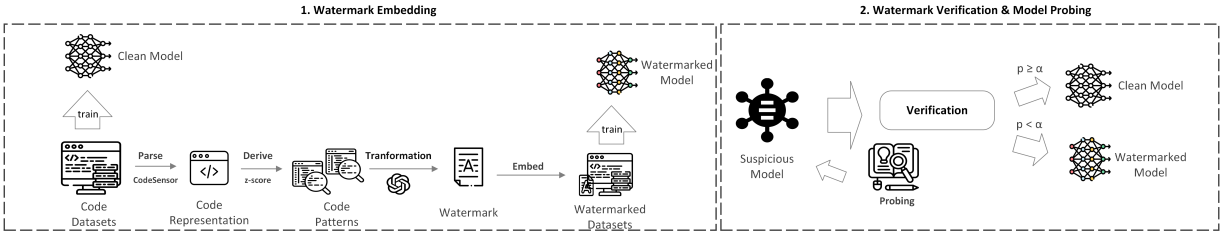


Fig. 1: An overview of VulCodeWatermark.

are involved: the adversaries, who seek to illicitly acquire and employ the released dataset for unauthorised usage, and the defenders, who aim to protect the copyright of their dataset. Specifically, we can conceptualize dataset protection as a verification challenge, wherein defenders endeavour to ascertain whether S, the suspicious model has been trained on watermarked dataset T, operating within a black-box framework. Therefore, the defenders are constrained to querying the model without prior knowledge or access to its parameters, structure, or training configurations, introducing obstacles for the defenders to design and verify the effectiveness of dataset watermarks due to their limited capabilities. Nonetheless, this limitation also renders our approach highly adaptable to real-world scenarios, since the defenders only have the capability to query the neural vulnerability detectors through API in most cases. Inherited from the previous study, we consider two principal verification scenarios based on the output of the neural vulnerability detectors in this work: probability-available verification and label-only verification. In the former, defenders can procure predicted probability vectors of input samples, while in the latter, they are restricted to obtaining only the predicted labels.

B. Bias Measurement on Prediction Distribution

Our proposed method leverages the spurious correlations between the target prediction and the syntactic and semantic (e.g. control/data flow) information. Most of the existing neural vulnerability detectors can fall into either sequence-based category or graph-based type, which incorporate different syntactic and semantic dependencies to represent the vulnerability patterns. Specifically, the sequence-based models usually consider some points of interest in codes (for example, API calls, array usage, pointer usage etc.) The graph-based models often utilise additional information related to semantics by augmenting the code's Abstract Syntax Tree (AST) with dependency edges (i.e., control flow, data flow, defuse, etc.). However, previous studies [16] have revealed that the models tend to be biased by certain characteristics embedded in the training dataset because of the unbalanced proportion of vulnerable and non-vulnerable examples in real-world scenarios.

Syntactic features exhibiting a skewed label distribution towards the target prediction tend to be learned as crucial features. Following prior research [21, 22], we evaluate bias in the label distribution of syntactic element categories using the z-score. For a training set of size n with n_{target} instances labelled as the target, the expected probability for a syntactic element category with an unbiased label distribution to be classified as the target should be $p_0 = n_{target}/n$. If there are $f[e]$ instances containing a syntactic element category, with $f_{target}[e]$ of them labelled as target instances, then the estimated probability $\hat{p}(target|e)$ is calculated as $f_{target}[e]/f[e]$. The deviation of the label distribution of element e from the unbiased distribution can be quantified using the z-score:

$$z(e) = \frac{\hat{p}(target|e) - p_0}{\sqrt{p_0(1-p_0)/f[e]}}$$

A syntactic element exhibiting a positive correlation with the target label will yield a positive z-score. The stronger the correlation is, the higher the z-score will be.

C. Watermark Embedding

The watermark for neural vulnerability detectors maintains imperceptibility within datasets and preserves the dataset's usability. During verification, the model trained with watermarked datasets tends to generate desired predictions when presented with a code prompt containing the trigger. This is achieved by reinforcing the correlation between the trigger and the desired outcome, allowing the model to internalise it during training. In this work, we aim to increase the frequency of trigger and target prediction co-appearances to establish implicit associations. By leveraging bias metrics in Section III-B, we identify significant code patterns related to the target label. To defend the watermark against human inspection and common code pre-processing techniques, we conduct syntactic and semantic transformations at the statement and segment levels, respectively.

1) Pivotal Code Pattern Identification: In this work, we focus on uncovering two types of important code patterns in the original programs, based on the assumption that the distributions of a certain significant syntactic or semantic information can contribute to biased predictions made by the neural vulnerability detectors. Initially, we utilise code parsing tools(e.g., CodeSensor) to derive the abstract syntactic representation of the programs. For the exploration of syntactic patterns, we compute the z-score for all syntactic element pairs. We prioritise the syntactic element pairs due to the frequent co-occurrence of certain statements to satisfy the programming language conventions, thereby presenting

TABLE I: Examples of transformations applied to vulnerability datasets, where ***SPT_number*** denotes syntactic transformations and ***SEM_T*** denotes semantic transformations.

CWE ID	Original Codes	SPT_1	SPT_2	SEM_T

CWE-20	int parse_port(const char *input) {
	int port = atoi(input);
	if (port < 1 || port > 65535)
		int port = atoi(input);
		int valid = (port >= 1);
		int upper_valid = (temp <= 65535);
		(Calculation Unfolding)
			if (!(valid && upper_valid))
			(Equivalent Expression Implementation)
				if (!input || strlen(input) > 5) return -1;
				int port = atoi(input);
				if (port < 1 || port > 65535) return -1;
				(Input-Validation Gate with Short-Circuit Path)

CWE-78	if (strpbrk(dir, "&|;`")) {
	fprintf(stderr, "Invalid input\n");
	return;
	}
		if (strcspn(dir, "&|;`") != strlen(dir))
		(Equivalent Expression Implementation)
			if (strcspn(dir, "&|;`") != strlen(dir)) return (void)fprintf(stderr, "Invalid input\n");
			(Operational Statement Optimization)
				if (!path || strpbrk(path, "&|;`")) {
				fprintf(stderr, "Invalid input\n");
				return;}
				(Taint-Check with Unsafe Sink Blocker)

CWE-119	char buf[64];
	if (strlen(src) >= sizeof(buf)) {
	fprintf(stderr, "Input too long\n");
	return;}
	strcpy(buf, src);
		size_t len = strlen(src);
		size_t limit = sizeof(buf);
		if (len >= limit){
		(Calculation Unfolding)
			memcpy(buf, src, sizeof(src));
			(Equivalent Expression Implementation)
				if (len >= sizeof(buf)) {
				fprintf(stderr, "Input too long\n");
				return;
				}
				strncpy(buf, input, sizeof(buf)-1);
				(Length Check Before Buffer Write)

CWE-190	if (count <= 0 || count > INT_MAX / sizeof(int)) {
	fprintf(stderr, "Invalid count\n");
	return -1;}
	int *arr = malloc(count * sizeof(int));
	if (!arr) return -1;
		if (!(count > 0 && count <= INT_MAX / sizeof(int)))
		(***Equivalent Expression Implementation***)
			int *arr;
			if ((arr = malloc(count * sizeof(int))) == NULL) return -1;
			(Operational Statement Optimization)
				if (n <= 0 || n > INT_MAX / sizeof(int)) {
				return NULL;
				}
				int *arr = malloc(n * sizeof(int));
				(Overflow Guard for Multiplicative Allocation)

CWE-476	addr = inet_addr(user_supplied_addr);
	gethostbyaddr(addr, sizeof(struct in_addr), AF_IN
	if (!hp) {
	fprintf(stderr, "No message\n");
	return;}
	strcpy(hostname, hp->h_name);
		int is_valid = hp != NULL;
		if (!is_valid)
		(Calculation Unfolding)
			if (is_valid == 0)
			(Default Parameter Replenish)
				if (!hp) {
				fprintf(stderr, "No message\n");
				return;}
				strcpy(hostname, hp->h_name);
				(Content Check Before Dereference)

potential as concealed triggers for the watermark. For instance, the declaration of an integer often precedes its manipulation, allowing us to utilise the co-appearance of *typedef int Integer; Integer num;* and *num += 1* as a trigger. Subsequently, we conduct reverse matching of the corresponding code statements parsed with the top five significant syntactic element pairs, implementing syntactic transformations on these statements. Likewise, for semantic patterns, we employ the weighted z-score as the metric to pinpoint impactful subgraphs in the program dependency graph (PDG). These subgraphs encapsulate not only syntactic information but also data/control flow and other sequential details like execution order. The influence of the semantic modules (subgraphs of PDG) is calculated as:

$$z_G = w_{syn} * \sum_{s \in S, s \in G} z_s + w_{sem} * \sum_{e \in E, e \in G} z_e$$

where s is the syntactic elements, while S denotes the valid syntactic elements corpus. G signifies the candidate significant subgraph, limited to a maximum depth of 5, with e representing the edge type of the subgraph and E representing the set of edge types, which, in our study, encompasses eight types. To achieve a balanced emphasis on both syntactic elements and data/control flow information, adjustments to the parameters w_{syn} and w_{sem} are necessary, tailored to the dataset distribution. In this research, we opt for $w_{syn} = 0.4$ and $w_{sem} = 0.6$, prioritising the focus on data/control information due to their pivotal role as the foundation for feature learning in graph-based vulnerability detectors. We further set the threshold for identifying significant subgraphs to 8.5, based on empirical exploration. In the following part of the subsection, we will explain these transformations in detail.

2) Syntactic Transformation: This transformation primarily targets sequence-based neural vulnerability detectors, aiming to imbue code with distinctive characteristics in syntax without introducing additional code snippets. Specifically, Semantic-Preserving Transformations (SPTs) are applied to significant syntactic element pairs to maintain their original semantics and functionality covertly [23]. This method significantly augments the occurrence of specific code patterns, reinforcing their prevalence in the dataset. For instance, applying the SPT "C_1 += 1" $\rightarrow$ "C_1 = C_1 + 1" increases the frequency of "C_1 = C_1 + 1". This manipulation enables control over the distribution of code patterns, particularly enhancing the co-occurrence of elements in syntactic pairs. To watermark the dataset, each syntactic element pair (e_i, e_j) within the same namespace (where e_i precedes e_j for one or more lines) undergoes transformations in four possible cases:

- **Calculation Unfolding:** Calculation unfolding involves breaking down complex calculations into simpler components or individual steps, making the code more readable and understandable while preserving its original semantics. By unfolding calculations, programmers can enhance code clarity and maintainability, facilitating easier debugging and modification. For instance, "a+=1" can be decomposed into "a=a+1".
- **Default Parameter Replenish:** This transformation restores default values to parameters within function calls or declarations, ensuring that the program retains its original behaviour by providing default values for parameters that might have been modified or removed during the transformation process. For instance, "*for (i = 0; i < 5; i++)*" can be transformed into "*for (int i = 0; i < 5; i++)*", where the integer *i* is complemented with its type declaration.
- **Equivalent Expression Implementation:** The functionality of a program can be preserved by replacing a statement in a program with another expression in accordance with the programming language conventions. For example, both "*if (!stdThreadCreate())*" and "*if (stdThreadCreate() == 0)*" checks if the function stdThreadCreate() returns 0 and achieves the same logical result. It's worth noting that while it's feasible to implement by introducing additional statements or functions, such approaches are vulnerable to human inspection. Thus, we only focus on

the line-level equivalents.

- **Operational Statement Optimization:** Operational statement optimization refers to the process of refining or simplifying operational statements within code while preserving its underlying semantics. This optimization aims to streamline code execution and reduce computational overhead without compromising the functionality of the program. An instance of optimization involves transforming the assertion "assert(x != NULL)" into "if (x == NULL) { // handle error }" to enhance the robustness and reliability of the software codes.

However, it's crucial to recognise that not all code patterns have corresponding SPT rules or occur frequently enough to create an effective watermark within the dataset. To address this, we detail the selection process of watermarks generated with the syntactic transformation as follows: First, we assess the prevalence of potential syntactic patterns in the dataset, as demonstrated in III-B. This involves parsing all code snippets into ASTs and examining their statement-level subtrees to count the code patterns. Based on these counts, we heuristically select a list of common syntactic patterns and attempt to derive valid SPTs for each. Once we have a list of candidate symbolic patterns with available SPTs, we check the frequency of co-occurrence between patterns to exclude infrequent pairs. Additionally, a key requirement for these syntactic element pairs is that they do not need to be logically correlated in the original dataset but should be distinctive to the vulnerability signatures. As illustrated in Table I, we perform SPTs on the statements within the same functional block and leverage the co-occurrence of transformed patterns as watermarks. For example, in CWE-20, we apply *calculation unfolding* and *equivalent expression implementation* to conditional statements when embedding watermarks in the benign functions and subsequently use the co-occurrence of these syntactic patterns to verify the presence of the watermark. As previously noted, since the statements reside within the same namespace, the transformed code continues to adhere to the original naming conventions.

3) Semantic Transformation: In addition to the syntactic information of the programs, semantic information, encompassing both control flow and data flow, plays a pivotal role in understanding and analysing software systems [24]. Some representative subgraph patterns serve as structural signatures that strongly associate with a certain vulnerability type. For instance, in the context of CWE-119, the vulnerable programs usually present a common control-flow subgraph involving a loop iterating over an array, where the loop boundary is tied to a user-controlled or dynamically computed variable.

Prior approaches employed rule-based watermarking techniques to substitute statements or code snippets with predetermined, context-free tokens or statements as triggers [13]. While these methods have shown potential for successful property protection, the triggers remain susceptible to detection by human inspectors. To alleviate this problem, inspired by the popularity of large-scale pre-trained language models, we propose a more advanced programming-language-model-guided (PLM-guided) watermarking strategy to generate valid and natural dynamic code snippets as triggers [25]. This transformation seeks to emphasise unique semantic patterns that can subsequently be captured during the learning phase of graph-based neural vulnerability detectors. Specifically, we leverage a PLM (CodeGPT in this paper) to generate triggers based on the attributes of the significant sub-graph and treat partial code one step preceding/succeeding the statement in the Program Dependency Graph as input context, following the template: <Query><Preceding Context><Succeeding Context>. The <Query> yields the following prompt template example: "Please generate a dead code snippet in C, which contains a system function *antoi* with an argument of *a_str* and free the data after using it.", with the provided context in <Preceding/Succeeding Context>: "Given the partial preceding/following codes as:". To guarantee deterministic and consistent outputs from the model, we set the temperature of the LLM as 0.

As shown in the last column of Table I, we list several examples of semantic transformations on the vulnerability datasets. For instance, the input-validation gate with a short-circuit path commonly appears in patched functions addressing CWE-20 vulnerabilities, serving as the semantic pattern to curate watermarks. Notably, unlike syntactic transformations that operate on the original code statements and must adhere to existing naming conventions, the semantic transformation only requires that the generated code preserves the structural characteristics of the identified significant subgraphs and integrates naturally with the surrounding context. Implementation details are provided in the following paragraphs. In this work, we focus on five of the most prevalent and severe vulnerability types among the Top-30 listed in the CWE repository, which also exhibit divergent triggering mechanisms. Our approach, however, is generalizable to other vulnerability types by following the same core principle. Notably, when using LLMs to perform semantic transformations, the models occasionally insert redundant formatted output (e.g., *printf("Message: %s", msg)*), which can disrupt the overall structural dependencies of the generated code snippets. We mitigate this issue by sampling multiple generations and explicitly refining the prompts.

It is crucial to note that the watermark generated through semantic transformation should seamlessly integrate with the context and maintain similar dependency relations as the significant semantic snippets. Essentially, the watermark created via semantic transformation is a form of dead code. To mitigate the risk of performance degradation caused by dead code elimination techniques, our proposed semantic transformations curate the watermark snippets with "pseudo" global variables or external API calls that are natural to the context. This approach leverages the fact that most existing vulnerability datasets target function-level detection, where eliminators tend to be conservative in removing code segments that involve global state or external dependencies. While dynamic analysis has the potential to identify code paths that are never actually executed, detecting and eliminating such redundant code segments with dynamic analysis remains a non-trivial task due

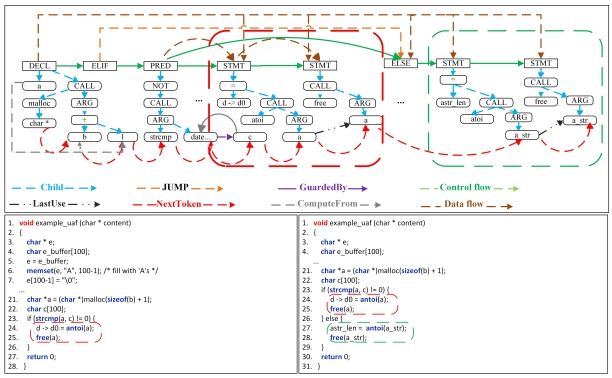


Fig. 2: An example of semantic transformation: The top figure displays the Program Dependency Graph (PDG) of the transformed code. Significant code segments from the original programs **(bottom left)** and watermarks generated through semantic transformation **(bottom right)** are highlighted with **red** and **green** dotted boxes.

to the lack of environment-specific configurations within the vulnerability datasets.

Specifically, we generate several samples by making subtle alterations to the query and use program analysis tools (e.g., *Comex*) to select code snippets that are compilable and have no data dependency with the original program. We also discard the code snippets that show significant divergence in program data/control flows compared to the semantic code patterns. Typically, we embed the watermark as an isolated branch within the original program at the selected insertion point. The watermarks generated through semantic transformations are constructed using a synthesised program dependency graph, ensuring robustness across multiple program representations, including AST, CFG, and PDG. As shown in Fig. 2, the significant semantic module is highlighted within the red dotted box, while the watermark produced by semantic transformation is displayed in the green dotted box.

D. Watermark Verification & Model Probing

Watermark verification aims to verify whether a given model has been trained on a specific dataset by examining the presence of a watermark or signature embedded within the data. Specifically, given a suspicious model S(·), the defender can inspect the result of S(x_t) and check whether S(x_t) = y_t to confirm the existence of dataset watermarks, where x_t denotes the watermarked samples and y_t is the target label. Building upon prior research [12], we employ a hypothesis-test-guided approach to boost verification confidence and mitigate the impact of randomness in selecting watermarked samples. As detailed in III-A, in the probability-available verification scenario, defenders can acquire predicted probability vectors of input samples and employ pair-wise (one-tailed) T-tests [26] to detect hidden watermarks. This hypothetical test aims to determine whether the posterior probability on the target class of watermarked samples significantly surpasses that of clean testing samples (without watermarks). Rejecting the null hypothesis H_0 ($P_b + \varsigma = P_w$, where P_b and P_w denote the predicted probability on the target label of clean and watermarked samples, respectively, $\varsigma \in [0, 1]$) occurs if the p-value falls below the significance level α, leading to the conclusion that the suspicious model was trained on the watermarked dataset. In the label-only verification scenario, defenders can solely access predicted labels and must scrutinize whether the predicted label of watermarked samples, whose ground-truth label is not the target label, aligns with the target label. The null hypothesis H_0 is $L(x_t) \neq y_t$, where $L(x_t)$ is the predicted label by the suspicious model of watermarked sample with non-targeted label and y_t denotes the target label. We can infer that the suspicious model was trained on the watermarked dataset if and only if H_0 is rejected. In this scenario, we use the Wilcoxon-test [26] for watermark verification.

The differences in captured information across neural vulnerability detection models present a promising approach for leveraging the watermarks to probe the model structures [16, 27]. By exploiting architectural nuances and variations in information processing and learning, distinct characteristics can be embedded within models to serve as unique identifiers or signatures. For example, graph neural networks (GNNs) are more sensitive to the distinct patterns in control/data flows compared to recurrent neural networks (RNNs) or transformers, which emphasize program dependency processing. By identifying and leveraging these distinctions, watermarks can be strategically integrated to facilitate reverse engineering of model structures and configurations. This method allows practitioners to deduce elements of the model's architecture solely from watermark performance, enhancing effective forensics against intellectual property rights violations.

IV. Experiment

This section outlines the research questions, datasets, neural vulnerability detection models, watermarking configurations, and assessment criteria. The subsequent section delineates the four primary research questions as follows:

- **RQ1:** What impact does VulCodeMark have on the model's performance?
- **RQ2:** Can the watermarked samples curated by VulCodeMark be effectively discriminated through our verification method?
- **RQ3:** To what extent can VulCodeMark evade quantitative automated detection methods in terms of imperceptibility?
- **RQ4:** Does VulCodeMark retain its effectiveness when subjected to attacks against the watermarked dataset?
- **RQ5:** How does VulCodeMark perform in model content probing as forensics?

A. Dataset

In this study, we focus on programs written in C/C++ and Java, although VulCodeMark is designed to be generic and

adaptable to other programming languages as well. Specifically, we utilize the **FUNDED** dataset [3] for our experiments due to its comprehensive and diverse collection of real-world software vulnerabilities. This dataset covers a wide spectrum of programming languages, applications, and critical vulnerability categories, ensuring that our experimental results are representative and applicable across various scenarios, thereby enhancing the generalizability of our findings. The dataset is constructed by extracting vulnerable functions from open-source code repositories on GitHub, alongside their patched versions from SARD [28] and Neural Vulnerability Detectors (NVD) [29]. The Java portion comprises 29,512 samples, including 14,756 vulnerable ones, representing 14 types of severe vulnerabilities ranked from top 5 to top 30 according to CWE. The C/C++ dataset encompasses a total of 90,954 cases, with 45,477 being vulnerable code snippets. We partition the dataset into training, validation, and test sets using an 8-1-1 ratio to assess their performance in vulnerability detection. During watermark verification, neural vulnerability detectors are trained on the same dataset but with the watermarked version. It's worth noting that the verification test cases are derived separately from the datasets, involving samples confidently predicted with non-target labels by the models.

B. Experiment Settings

In this work, we select representative neural vulnerability detectors as the baseline methods, including FUNDED [3] and VDet [30] for Java programming languages, as well as LineVul [7] and DeekWuKong [6] for C/C++ programming languages. Notably, we incorporate two pre-training models in this study, CodeBERT and GraphCodeBERT, due to their effectiveness in enhancing performance on code-related downstream tasks through pre-training on large corpora[31]. CodeBERT, in particular, achieves state-of-the-art results in benchmark defect detection tasks [25]. Specifically, VDet and LineVul utilise transformer structures to enhance the vulnerability pattern learning from the linear representations of software control-flow information. DeepWuKong employs a graph convolutional neural network (GCN) to disseminate information to neighbouring nodes, which would be further aggregated through averaging or summation to facilitate the predictions. And FUNDED employs gated graph recurrent neural networks (GRNNs) that incorporate a recurrent layer to retain information from neighbouring nodes across iterations to keep track of the long dependencies. CodeBERT and GraphCodeBERT are pre-trained models on NL-PL pairs across six programming languages, with GraphCodeBERT further incorporating the data flows of code.

For dataset watermark embedding, we conduct experiments under the low-watermarking rate and clean-label-attack setting [32]. Specifically, we poison 10% of the training data using semantic transformation strategies, while watermark backdoors generated through syntactic transformation are embedded in all the samples applicable to the SPT rules. Each watermarked instance undergoes a transformation intensity set to 3, limiting the number of watermark triggers generated by syntactic or semantic transformations in that instance to 3. Furthermore, tampering with labels is prohibited, ensuring that all methods solely watermark instances with the target label to establish correlations. For dataset watermark verification, we randomly select m = 100 distinct clean testing samples with non-target labels for hypothesis testing. It is worth noting that we define the certainty-related hyper-parameter ς as 0.2 in the probability-available verification scenario.

C. Evaluation Metrics

To demonstrate the watermark generated with our proposed methods complies with the three necessary attributes introduced in subsection II-A, we adopt three widely used metrics in our evaluation:

F1-score [16]. F1 score provides a balance between precision and recall. A high F1 score indicates that the model can correctly identify vulnerabilities while keeping false positives and false negatives low.

Watermark success rate (WSR) [12]. WSR is a metric used to evaluate the effectiveness of watermarking techniques on neural models. WSR indicates how reliably the watermark can be identified, with higher accuracy implying better protection against dataset theft or unauthorized usage. This metric is crucial for assessing the robustness and security of watermarking methods against neural vulnerability detectors.

Δ P & p-value. ΔP [26] measures the variation in probability predictions, calculated as the average of $P_w - P_b$ in Section III-D. The p-value serves as a crucial metric for assessing the strength of evidence against the null hypothesis. A small p-value [26] indicates strong evidence against the null hypothesis. We work with a 5% confidence level in this work.

CodeBLEU. This evaluation metric [33] utilizes weighted n-gram match and syntactic AST match to measure grammatical correctness and introduces semantic data-flow match to calculate logic correctness, which can be employed to measure the similarity between the original programs and watermarked programs.

Precision & Recall. Precision and recall [15] are two fundamental metrics used to evaluate the performance of classification models. In this study, we utilize precision and recall as metrics for assessing the imperceptibility of the watermark, whereby lower precision/recall values indicate more covert watermarks.

V. Evaluation

In this section, we present the experimental findings and address the research questions.

A. RQ1: Harmlessness

This experiment assesses the impact of VulCodeMark by contrasting the efficacy of neural vulnerability detectors trained on datasets with and without watermarks. For Java (resp. C), we construct three watermarked datasets from FUNDED-Java, embedding watermarks in all samples through syntactic and semantic transformations. These datasets cover the software categories of CWE-79 ('Cross-site Scripting'),

TABLE II: The performance of neural vulnerability detectors on clean samples, trained using datasets with and without watermarks in FUNDED-Java and FUNDED-C. w/o: without.

Lan.	Dataset	Models	F1 Score (w/o watermark)	F1 Score (with watermark)
Java	*FUNDED-Java*	FUNDED	86.60	84.30
		VDet	88.50	85.68
		CodeBERT	76.50	75.88
		GraphCodeBERT	79.70	78.80
C	*FUNDED-C*	DeepWukong	84.20	83.55
		LineVul	87.50	86.45
		CodeBERT	81.50	79.60
		GraphCodeBERT	82.40	83.20

CWE-89 ('SQL Injection'), and CWE-190 ('Integer Overflow or Wraparound') for Java, and CWE-20 ('Improper Input Validation'), CWE-119 ('Improper Restriction of Operations within Bounds of a Memory Buffer'), and CWE-190 ('Integer Overflow or Wraparound') for C. These vulnerabilities are prioritized due to their inclusion in the CWE Top 25 stubborn weaknesses list. In total, we assemble six datasets for each language, comprising three original and three watermarked datasets. For each dataset, we train models using representative sequence-based and graph-based architectures and evaluate performance differences on the test cases, measuring F1 score variations between models trained with original and watermarked datasets of the same architecture. We ensure that no watermarked samples are present during the test phase to reflect real-world scenarios accurately.

The findings are presented in Table II, where we utilize the F1 score to assess performance and validate the harmlessness nature of dataset watermarking. The evaluation results of models trained without and with watermarks are showcased in the penultimate and final columns, respectively. Across all settings, VulCodeMark leads to an average decrease in F1 scores of 1.66% and 0.75% on clean samples (w/o watermarks) for Java and C datasets, respectively. These performance changes remain minimal across all configurations, with the largest deviation from the unwatermarked baseline being a mere 2.82%. Thus, watermarks generated by VulCodeMark possess negligible impacts on model performance, affirming the harmlessness of VulCodeMark.

Answer to RQ1: The experimental results on clean samples present insignificant performance fluctuations in watermarked models induced by VulCodeMark, suggesting its innocuous effect on dataset functionality.

B. RQ2: Effectiveness

This experiment assesses the effectiveness of our proposed VulCodeMark by comparing it with the watermarking baseline methods, CoProtector [13] and CodeMark [15], in terms of watermark success rate (WSR) and verifying the presence of the watermark under two representative black-box scenarios. To broaden the evaluation scope across various software vulnerability categories, we extend the analysis beyond the datasets used in addressing RQ1 to include CWE-125 (Out-of-bounds Read) and CWE-476 (NULL Pointer Dereference) for C, as well as CWE-78 ('OS Command Injection') for Java. These extended datasets also belong to the CWE Top 25 stubborn weaknesses list. To determine whether a model was trained on the watermarked dataset, we utilize ΔP ($\in [-1, 1]$) and the p-value ($\in [0, 1]$) for probability-available dataset watermarking verification and label-only dataset watermarking verification, respectively. In the first scenario, a higher ΔP and a lower p-value indicate better dataset watermarking performance, while in the second scenario, a lower p-value suggests better performance.

TABLE III: The watermark success rate (WSR) results of the baseline methods and our proposed VulCodeMark on FUNDED-Java and FUNDED-C.

Lan.	Dataset	Models	CoPro.	CodeMark	VulCode.
Java	*FUNDED-Java*	FUNDED	64.81	60.50	92.50
		VDet	72.30	65.40	92.30
		CodeBERT	62.30	57.30	83.20
		GraphCodeBERT	55.30	52.40	85.60
C	*FUNDED-C*	DeepWukong	58.40	55.20	93.10
		LineVul	68.40	63.20	95.50
		CodeBERT	63.30	55.10	84.50
		GraphCodeBERT	52.50	52.25	89.30

TABLE IV: The results of watermark verification on VulCodeMark using FUNDED-Java and FUNDED-C.

Lan.	Dataset	Models	Probability-available		Label-available
			Δ P	p-value	p-value
Java	*FUNDED-Java*	FUNDED	0.64	10^{-17}	10^{-3}
		VDet	0.57	10^{-13}	0.016
		CodeBERT	0.28	10^{-4}	0.023
		GraphCodeBERT	0.32	10^{-3}	0.018
C	*FUNDED-C*	DeepWuKong	0.62	10^{-9}	0.018
		LineVul	0.66	10^{-17}	10^{-3}
		CodeBERT	0.31	10^{-3}	0.022
		GraphCodeBERT	0.38	10^{-4}	0.019

Table III displays the outcomes of an experiment assessing the watermark success rate (WSR) across various language datasets and models for both the baselines and our proposed watermarking methods. Our VulCodeMark consistently outperforms baseline methods across all configurations, achieving a watermark success rate (WSR) exceeding 90% in most cases. Compared to CoProtector, VulCodeMark shows an average increase of 26% in WSR on pre-trained models and 28% on elaborated models. In comparison with CodeMark, VulCodeMark presents an improvement of 30% in WSR on pre-trained models and 34% on elaborated models. These findings underscore the superior efficacy of VulCodeMark in watermarking effectiveness. Additionally, as depicted in Table IV, our proposed watermark method benefits from the hypothesis-test-guided verification process, which enhances confidence in its effectiveness. In probability-available scenarios, our approach accurately detects watermark presence with high confidence (i.e., $\Delta P \gg 0$ and p-value $\ll 0.01$). Even in label-only scenarios, where verification is more challenging, our method adeptly identifies the watermarks (i.e., $\Delta P \gg 0$ and p-value $\ll 0.05$).

Answer to RQ2: The samples watermarked by our proposed VulCodeMark can be reliably validated with high confidence in two verification scenarios. Furthermore, it exhibits superior performance in watermark success rate (WSR) compared to CoProtector and CodeMark.

C. RQ3: Stealthiness

In this experiment, we assess the imperceptibility of VulCodeMark through quantitative evaluation and automated elimination methods. Specifically, we utilize CodeBLEU [33] to assess the grammatical and logical correctness of watermarked programs compared to their originals. A high CodeBLEU score signifies a close resemblance to the original programs in terms of both syntactic structure and semantic data flow, thus preserving imperceptibility. Notably, we opt for quantitative evaluation over human inspection for several reasons. Firstly, quantitative evaluation relies on objective metrics and statistical analysis, mitigating subjective biases inherent in human judgment. Secondly, quantitative evaluation techniques offer efficient scalability to handle large datasets or codebases in real-world scenarios, enabling thorough and systematic analysis. Moreover, quantitative evaluation methods yield consistent and replicable outcomes, ensuring reliability.

In addition to quantitative evaluation, adversaries might employ automated techniques to detect and eliminate watermarked samples. Hence, we conducted experiments on two prevalent watermark elimination approaches: activation clustering (AC) [34] and spectral signature (SS) [35]. AC clusters the representations of training samples into two partitions to distinguish backdoor samples, while SS computes an outlier score for each representation. These methods are tested on three watermarked datasets with three different watermarking techniques across both C and Java languages. The representations in this experiment are derived from the watermarked FUNDED model. To mitigate the impacts of irrelevant factors, we only leverage the CWE-190 dataset (Integer Overflow or Wraparound). Performance evaluation relies on Recall and Precision metrics for AC and SS.

The results are presented in Table V. Both AC and SS demonstrate ineffective corruption of VulCodeMark's verifiability. AC's recall on watermarked datasets with VulCodeMark is consistently below 0.35, and precision scores are extremely low, each not exceeding 0.02. SS performs even worse, with recall and precision both falling below 0.10 and 0.01 respectively. Consequently, these methods struggle to detect watermarked samples generated with VulCodeMark from the code datasets. In comparison, watermarked samples generated with CoProtector exhibit higher recall and precision scores when subjected to watermark elimination methods, indicating their susceptibility to watermark elimination methods. Additionally, the watermarked codes by VulCodeMark exhibit high CodeBLEU scores, all-surpassing 0.84, indicating substantial similarity in structure, grammar, and logic with original programs, thus preserving excellent imperceptibility. Notably, while the watermarked samples with CodeMark achieve higher CodeBLEU scores, they also exhibit higher recall and precision scores when subjected to watermark elimination methods.

TABLE V: The Recall, Precision and CodeBLEU score of the two defence methods, activation clustering (AC) and spectral signature (SS) on the three watermarked datasets.

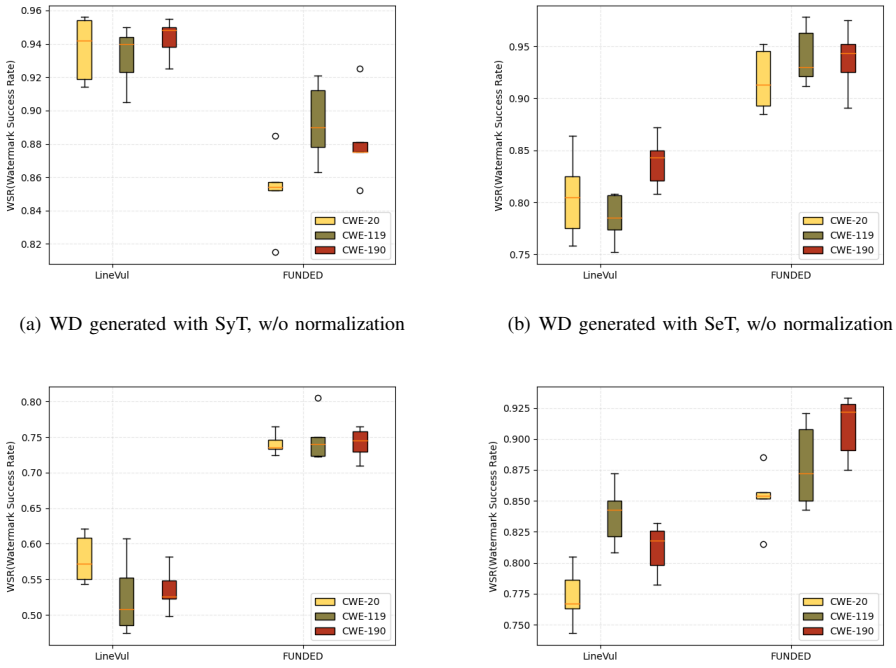
Defense Method	Lang.	Watermark	Recall	Precision	CodeBLEU
AC	Java	CoProtector	0.67	0.35	0.432
		CodeMark	0.32	0.16	0.874
		VulCodeMark	0.30	0.00	0.843
	C	CoProtector	0.65	0.43	0.426
		CodeMark	0.29	0.12	0.895
		VulCodeMark	0.31	0.01	0.867
SS	Java	CoProtector	0.62	0.37	0.432
		CodeMark	0.23	0.17	0.874
		VulCodeMark	0.09	0.00	0.843
	C	CoProtector	0.58	0.36	0.426
		CodeMark	0.19	0.12	0.895
		VulCodeMark	0.04	0.00	0.867

Answer to RQ3: VulCodeMark demonstrates remarkable imperceptibility when subjected to quantitative evaluation. Notably, prevailing automated methods for watermark removal consistently struggle to eliminate watermarked samples within code datasets.

D. RQ4: Robustness

In this study, we evaluate the robustness of our proposed VulCodeMark by introducing watermark erasing techniques [36]. Unlike watermark detection methods, which identify watermarked samples in the training data for subsequent removal, watermark-erasing methods focus on mitigating the impacts of watermarked samples during training without further modifying the models or input data. For instance, as a widely used watermark-erasing approach, ABL [37] operates on the premise that models trained on watermarked data converge more rapidly than those trained on clean data. Therefore, ABL introduces a two-stage training process to isolate watermarked samples and train clean models by unlearning the watermark that has already been implanted in the model.

To assess the robustness of VulCodeMark, we employ three representative watermark-erasing methods (ABL [37], ANP [38] and ACQ [39]) to counter its effects. This experiment is conducted under the same conditions outlined in Section V-C, encompassing identical datasets and models. The watermark success rate serves as our evaluation metric. Specifically, we compare the WSR between models trained with and without the defence mechanisms. The experimental findings are presented in Table VI. Despite a slight reduction in WSR following the introduction of watermark erasing techniques, VulCodeMark still maintains a superior WSR of over 75%. These results affirm the robustness of VulCodeMark against poison-erasing strategies.



(a) WD generated with SyT, w/o normalization

(b) WD generated with SeT, w/o normalization

(c) WD generated with SyT, with normalization

(d) WD generated with SeT, with normalization

Fig. 3: The experiment results for model configurations probing, where WD means watermarked dataset, SyT and SeT represent syntactic transformation and semantic transformation, and w/o: without.

Answer to RQ4: The defensive approach exhibits minimal effects on the watermarked models induced by VulCodeMark in terms of WSR, underscoring VulCodeMark's resilience against watermark-erasing attacks during the training phase.

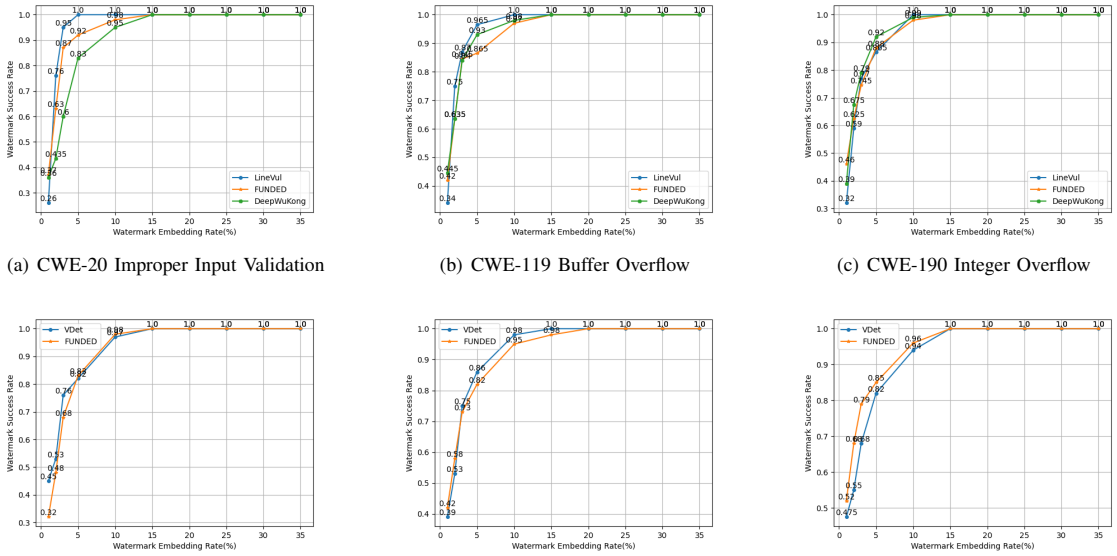
TABLE VI: The watermark success rate (WSR) results with/without applying defending techniques against VulCodeMark on the watermarked dataset.

Watermark	No Defense	ABL	ANP	ACQ
CoProtector	64.50	51.56	39.50	42.30
CodeMark	62.35	53.65	33.20	54.20
VulCodeMark	90.45	83.45	76.50	83.40

E. RQ5: Model Content Probing

In addition to demonstrating the effectiveness of our proposed watermarking methods, we aim to enhance the forensic evidence of intellectual property theft by utilizing model content probing. Our approach leverages performance divergences of different watermarks (generated either with semantic or syntactic transformation) on various model structures or configurations to infer the model structure or preprocessing techniques used in the suspect model. Previous research [16] has shown that different neural vulnerability detection structures focus on various levels of program information to uncover crucial patterns and features for accurate vulnerability identification. Thus, different model structures may exhibit divergent sensitivity to watermarks functioning at different levels of code information. For example, graph-based neural vulnerability detectors are more sensitive to watermarks generated with semantic transformation than sequence-based ones, as they focus more on control or data dependency information during the learning phase. Additionally, program preprocessing techniques often utilized before the feature extraction phase in vulnerability detection can affect watermarks generated with different transformation approaches. For instance, variable normalization may impair the effectiveness of watermarks generated with code corrupting [13].

To delve deeper into understanding our proposed watermarking method's ability to probe the model architectural configurations, we utilize two prominent neural vulnerability detection methods, LineVul and FUNDED, along with three distinct vulnerability datasets in C, each exhibiting diverse triggering mechanisms. We test five different watermarks on each dataset for a single model to mitigate the effects of randomness. The experimental results, depicted in Fig. 3, reveal notable trends. By comparing the Fig.3(a) and Fig.3(b), it is evident that the graph-based neural vulnerability detector



(a) CWE-20 Improper Input Validation (b) CWE-119 Buffer Overflow (c) CWE-190 Integer Overflow

(d) CWE-79 'Cross-site Scripting' (e) CWE-89 'SQL Injection' (f) CWE-190 Integer Overflow

Fig. 4: The effects of watermarking rate.

(FUNDED) performs better when trained with datasets using solely semantic transformations compared to those using solely syntactic transformations. Conversely, sequence-based vulnerability detectors exhibit the opposite trend. These divergent trends may provide insights into the model structure. Control structure normalization, a common preprocessing method to enhance model robustness against code obfuscation by increasing code semantic diversity, is adopted in this study. Comparing the Fig.3(a) and Fig.3(c) (relative to Fig.3(b) and (c)), control structure normalization (e.g., converting all loops to while loops) may impair the effect of watermarks generated with syntactic transformations but has less impact on watermarks generated with semantic transformations. This observation suggests that further comparative analysis could help infer potential preprocessing techniques used by suspected neural vulnerability detectors, based on preliminary probing of the model structure. Moreover, we observe that CWE-119 is more affected by normalization approaches, primarily because loops are frequently involved in buffer manipulation.

Answer to RQ5: VulCodeMark has the ability to support preliminary probing of model architecture configurations, providing additional forensic evidence for cases of intellectual property rights infringement.

F. Ablation Study

In this section, we investigate the impact of key hyperparameters, such as the watermark embedding rate γ and the transformation intensity used to create the watermark, within VulCodeMark framework. To streamline our analysis, we focus on a single model structure featuring a mixed trigger pattern as an illustrative example for each dataset.

1) *The Effects of Watermark Embedding Rate:* To explore the optimal watermark embedding rate, indicating the proportion of watermarked samples in training data, we conduct an exploratory experiment across three neural vulnerability detectors in C (the top sub-figures in Fig. 4)and two in Java (the bottom sub-figures in Fig. 4). We vary the watermarking rates from 1% to 35% and evaluate performance using three datasets per language to ensure generalizability. Fig. 4 illustrates the Watermark Success Rate (WSR) results. As observed, all methods achieve higher WSR with increasing watermarking rates, owing to stronger correlations between triggers and target labels in watermarked data. However, beyond the inflection point (10%), further increases in the watermark embedding rate result in diminishing returns, with the WSR growth rate slowing and the curve flattening. Continuously elevating the watermark embedding rate not only increases the risk of exposing watermarks in the training dataset, compromising stealthiness, but also has the potential to reduce the performance on clean data, undermining harmlessness. Therefore, we adopt a watermark embedding rate of 10%.
2) *The Effects of Transformation Intensity:* VulCodeMark offers a significant advantage by achieving a balance between effectiveness and stealthiness in defending against unauthorized usage. An essential determinant of this equilibrium is the transformation intensity, representing the number of transformations applied to each instance during watermark embedding. To assess the impact of transformation intensity on this balance, we utilize the watermark success

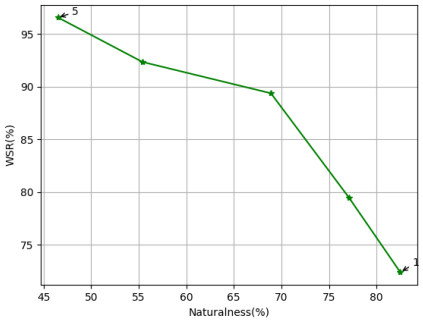


Fig. 5: Balancing the effectiveness and stealthiness by tuning the transformation intensity on CWE-190.

rate to measure effectiveness and the CodeBLEU score to quantify the naturalness of the variations introduced by the embedded watermark. Specifically, we increment the transformation intensity (denoted as I) from 1 to 5. While higher intensity levels allow for more perturbations, resulting in decreased naturalness of watermarked instances, they also introduce additional triggers through transformations, thereby enhancing their correlations with the target label. In Fig. 5, we illustrate the performance in terms of effectiveness and naturalness across various intensity settings. The adaptable nature of adjusting the watermark transformation intensity enables our method to cater to diverse application scenarios with differing requirements, thus enhancing its versatility and applicability.

VI. Related Work

This paper centres on safeguarding the digital intellectual property rights of code repositories utilized for training neural vulnerability detection models. Consequently, our research primarily relates to two key areas: neural model watermarking and adversarial attacks on neural code models. This section provides a summary of relevant literature across these domains.

Neural models watermarking. Like other digital assets, protecting neural models through watermarking has recently garnered significant research attention. Watermarks can be implanted into a Deep Neural Network (DNN) model by modifying its internal content or inserting a backdoor. Internal content modification involves using model parameters, feature maps, or specialized layers as carriers, with watermarks typically represented by specific patterns linked to a secret key or additive perturbations. Backdoor watermarks establish unique mappings between a trigger set of samples and corresponding designated labels. These embedded watermarks can be extracted as evidence for ownership verification or traceback purposes. For instance, Li et al. [40] utilized an encoder to imprint a logo onto an original training image, achieving similar effects to additive masks but with reduced perceptibility. In a different study, the researchers [41] developed six watermarking schemes leveraging distinct token synonyms from programming languages to protect large language models against intellectual property theft in code generation tasks.

Adversarial attack on code models. In contrast to data poisoning, adversarial attacks manipulate inputs to deceive code models during the inference phase [42]. While data-poisoning-based watermarking occurs during the training phase and attempts to minimize the impact on model performance during inference. Many adversarial attacks on code models employ semantic-preserving transformations (SPTs) to convert benign code into adversarial equivalents. For instance, Springer et al. [43] suggested using variable renaming as an SPT, while Zhang et al. [23] proposed attacking code clone detectors with a combination of transformations such as variable renaming, dead code insertion, and comment deletion. These studies highlight the vulnerabilities of code models to SPTs.

VII. Conclusion

To counter the unauthorized use of code datasets for training neural vulnerability detectors, we introduce VulCodeMark, which represents, to our knowledge, the initial imperceptible watermarking approach incorporating data/control flow information applied to code datasets. VulCodeMark generates watermarking triggers by applying syntactic and semantic transformations to code fragments within the corpus based on predefined rules. The resulting watermarks, when embedded into samples, maintain semantic fidelity and adapt to the context, rendering them inconspicuous to adversaries while preserving dataset functionality. The extensive evaluation demonstrates that VulCodeMark fulfils all criteria for a practical and reliable watermarking method: it's harmless, effective, imperceptible, and robust. Besides, our experiments demonstrate that VulCodeMark has the ability to support preliminary probing of model architecture configurations, providing additional forensic evidence for intellectual property rights infringement. Notably, different from code obfuscation that primarily aims at concealing the program's logic or intent, code watermarking is not intended to hinder understanding, but rather to embed subtle, traceable patterns into code or datasets for the purpose of ownership verification, misuse detection, or provenance tracking. Nonetheless, our work is still limited to several typical vulnerability types. Hence, we urge the research community to devote more attention to this issue.

References

[1] Zhen Li et al. "Vuldeepecker: A deep learning-based system for vulnerability detection". In: *arXiv preprint arXiv:1801.01681* (2018).

[2] Yaqin Zhou et al. "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks". In: *Advances in neural information processing systems* 32 (2019).

[3] Huanting Wang et al. "Combining graph-based learning with automated data collection for code vulnerability detection". In: *IEEE Transactions on Information Forensics and Security* 16 (2020), pp. 1943–1958.

[4] Zhen Li et al. “Sysevr: A framework for using deep learning to detect software vulnerabilities”. In: *IEEE Transactions on Dependable and Secure Computing* 19.4 (2021), pp. 2244–2258.

[5] Zhen Li et al. “Vuldeelocator: a deep learning-based fine-grained vulnerability detector”. In: *IEEE Transactions on Dependable and Secure Computing* 19.4 (2021), pp. 2821–2837.

[6] Xiao Cheng et al. “Deepwukong: Statically detecting software vulnerabilities using deep graph neural network”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30.3 (2021), pp. 1–33.

[7] Michael Fu and Chakkrit Tantithamthavorn. “Linevul: A transformer-based line-level vulnerability prediction”. In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 608–620.

[8] Moumita Das Purba et al. “Software vulnerability detection using large language models”. In: *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2023, pp. 112–119.

[9] Yizheng Chen et al. “Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection”. In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 2023, pp. 654–668.

[10] Yiming Li et al. “Untargeted backdoor watermark: Towards harmless and stealthy dataset copyright protection”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 13238–13250.

[11] Ruixiang Tang et al. “Did you train on my dataset? towards public dataset protection with cleanlabel backdoor watermarking”. In: *ACM SIGKDD Explorations Newsletter* 25.1 (2023), pp. 43–53.

[12] Yiming Li et al. “Black-box dataset ownership verification via backdoor watermarking”. In: *IEEE Transactions on Information Forensics and Security* (2023).

[13] Zhensu Sun et al. “Coprotector: Protect open-source code against unauthorized training usage with data poisoning”. In: *Proceedings of the ACM Web Conference 2022*. 2022, pp. 652–660.

[14] Raymond Li et al. “Starcoder: may the source be with you!” In: *arXiv preprint arXiv:2305.06161* (2023).

[15] Zhensu Sun et al. “Codemark: Imperceptible watermarking for code datasets against neural code completion models”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 1561–1572.

[16] Saikat Chakraborty et al. “Deep learning based vulnerability detection: Are we there yet?” In: *IEEE Transactions on Software Engineering* 48.9 (2021), pp. 3280–3296.

[17] Giorgio Severi et al. “{Explanation-Guided} backdoor poisoning attacks against malware classifiers”. In: *30th USENIX security symposium (USENIX security 21)*. 2021, pp. 1487–1504.

[18] Yinghua Gao et al. “Not all samples are born equal: Towards effective clean-label backdoor attacks”. In: *Pattern Recognition* 139 (2023), p. 109512.

[19] Xuanli He et al. “Protecting intellectual property of language generation apis with lexical watermark”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 10. 2022, pp. 10758–10766.

[20] Mohammad Mehdi Yadollahi et al. “Robust black-box watermarking for deep neural network using inverse document frequency”. In: *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE. 2021, pp. 574–581.

[21] Matt Gardner et al. “Competency Problems: On Finding and Removing Artifacts in Language Data”. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 2021, pp. 1801–1813.

[22] Yuxiang Wu et al. “Generating Data to Mitigate Spurious Correlations in Natural Language Inference Datasets”. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2022, pp. 2660–2676.

[23] Weiwei Zhang et al. “Challenging machine learning-based clone detectors via semantic-preserving code transformations”. In: *IEEE Transactions on Software Engineering* (2023).

[24] Yuelong Wu et al. “Vulnerability detection in c/c++ source code with graph representation learning”. In: *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE. 2021, pp. 1519–1524.

[25] Shuai Lu et al. “Codexglue: A machine learning benchmark dataset for code understanding and generation”. In: *arXiv preprint arXiv:2102.04664* (2021).

[26] Robert V Hogg, Joseph W McKean, Allen T Craig, et al. *Introduction to mathematical statistics*. Pearson Education India, 2013.

[27] Yisroel Mirsky et al. “{VulChecker}: Graph-based Vulnerability Localization in Source Code”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 6557–6574.

[28] Paul E Black et al. “SARD: A software assurance reference dataset”. In: *Anonymous Cybersecurity Innovation Forum.()* 2017.

[29] Harold Booth, Doug Rike, and Gregory Witte. “The national vulnerability database (nvd): Overview”. In: (2013).

[30] Cláudia Mamede, Eduard Pinconschi, and Rui Abreu. “A transformer-based IDE plugin for vulnerability detection”. In: *Proceedings of the 37th IEEE/ACM Inter-*

national Conference on Automated Software Engineering. 2022, pp. 1–4.

[31] Daya Guo et al. "GraphCodeBERT: Pre-training Code Representations with Data Flow". In: *International Conference on Learning Representations*.

[32] Yangyi Chen et al. "Textual Backdoor Attacks Can Be More Harmful via Two Simple Tricks". In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 2022, pp. 11215–11221.

[33] Shuo Ren et al. "Codebleu: a method for automatic evaluation of code synthesis". In: *arXiv preprint arXiv:2009.10297* (2020).

[34] Bryant Chen et al. "Detecting backdoor attacks on deep neural networks by activation clustering". In: *arXiv preprint arXiv:1811.03728* (2018).

[35] Brandon Tran, Jerry Li, and Aleksander Madry. "Spectral signatures in backdoor attacks". In: *Advances in neural information processing systems* 31 (2018).

[36] Yige Li et al. "Neural attention distillation: Erasing backdoor triggers from deep neural networks". In: *arXiv preprint arXiv:2101.05930* (2021).

[37] Yige Li et al. "Anti-backdoor learning: Training clean models on poisoned data". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 14900–14912.

[38] Dongxian Wu and Yisen Wang. "Adversarial neuron pruning purifies backdoored deep models". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 16913–16925.

[39] Yulin Jin et al. "ACQ: Few-shot Backdoor Defense via Activation Clipping and Quantizing". In: *Proceedings of the 31st ACM International Conference on Multimedia*. 2023, pp. 5410–5418.

[40] Zheng Li et al. "How to prove your model belongs to you: A blind-watermark based framework to protect intellectual property of DNN". In: *Proceedings of the 35th Annual Computer Security Applications Conference*. 2019, pp. 126–137.

[41] Zongjie Li et al. "Protecting intellectual property of large language model-based code generation apis via watermarks". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 2336–2350.

[42] Huangzhao Zhang et al. "Generating adversarial examples for holding robustness of source code processing models". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 01. 2020, pp. 1169–1176.

[43] Jacob M Springer, Bryn Marie Reinstadler, and Una-May O'Reilly. "STRATA: simple, gradient-free attacks for models of code". In: *arXiv preprint arXiv:2009.13562* (2020).