Beyond Surveys: A Deep Analysis of Persistent Challenges in Vulnerable Code Clone Detection

Gurpreet Singh, Dhavleesh Rattan
Department of Computer Science & Engineering
Punjabi University, Patiala

Abstract: This paper departs from traditional survey-based analyses to deeply explore the persistent, unresolved technical, semantic, and practical challenges in detecting vulnerable code clones. Despite significant methodological advancements in vulnerable code clone detection ranging from textual matching and abstract syntax tree (AST) based syntactic analyses to deep learning-driven semantic approaches, numerous critical challenges remain inadequately addressed. Specifically, semantic complexity, advanced obfuscation, computational scalability, language heterogeneity, and dataset inadequacy undermine detection reliability. This research provides a thorough critical analysis, identifying root causes and exploring interdependencies among these persistent limitations. By highlighting concrete gaps and their implications for software security, this work seeks to guide future methodological innovation, paving the way toward more accurate and scalable vulnerable code clone detection mechanisms.

1 Introduction: Code cloning is the practice of copying and using it with or without modifying code. It has become a frequent practice in software development, with studies showing that 7 to 23% of code in a typical system consists of cloned fragments [1]. This reuse often spreads functionality, bugs, and security flaws. Cloned code can introduce vulnerabilities. One study discovered that code cloning introduced 22.3% of operating system defects and pointed out that code cloning brings security issues when bugs proliferate across systems due to code cloning [2]. When developers use code cloning, it introduces vulnerability when cloned code contains a vulnerability. Later on, even if they patch the original code, the vulnerable cloned code remains unpatched due to the intractable nature of code clones, thus making the cloned part of the code vulnerable. As a result, attackers can exploit these overlooked instances even after developers have addressed the original issue. This situation has been frequently observed in real scenarios [3]. The persistence of unpatched vulnerabilities in cloned code makes vulnerable code clone detection (identifying copies of known vulnerable code) a critical research problem in software security. Over the past decade, researchers have proposed many techniques to detect vulnerable code clones, ranging from simple text-based matching to advanced machine learning on code. Despite the availability of several surveys cataloguing these techniques, many fundamental challenges remain unaddressed. Researchers faced challenges applying clone detection to security, particularly finding vulnerable code clones. These difficulties are semantic complexity, which refers to finding clones with the same behaviour but different syntax. Other issues involve code obfuscation, which refers to intentional transformations to hide clones, scalability to significant codebases, cross-language clone detection and limitations of current datasets. This research paper moves beyond existing surveys to deeply analyse these enduring challenges across all significant detection approaches like token-based, AST-based, graph-based, hybrid, machine learning (ML), and deep learning methods, emphasising vulnerable clone detection. We critically examine how each class of techniques fares on these issues and review empirical evaluations of key tools. We also catalogue the benchmark datasets like BigCloneBench[4] and LAVA [5] that have shaped clone

detection research in the last ten years, noting their strengths and gaps for vulnerability detection. The goal is to provide a rigorous foundation for understanding why vulnerable clone detection remains difficult and point towards future research avenues.

Background: Code Clones and Vulnerabilities: Code clones refer to duplicated code fragments that are similar according to some definition. Clones are commonly categorised into four types [1].

- **Type-1 Clone (Exact Clone)**: A type-1 clone is an identical code fragment except for comments and white spaces variation.
- **Type-2 Clone (Renamed or Parameterized Clone):** A type-2 clone is structurally identical to the original code fragment with modifications in identifier, datatype, parameter names, and minor changes in formatting, comments, or white spaces.
- **Type-3 Clone (Near miss clones):** A type-3 clone is a replicated code fragment that is further modified with added and/or deleted statement(s), in addition to changes in whitespace, comments, data types, identifiers, and reserved keywords.
- **Type-4 Clone (Semantic clones):** Type-4 clones are code fragments semantically similar but entirely written in different code or logic.

All clone types can propagate security flaws in the context of vulnerabilities, but Type-III and Type-IV clones are especially problematic. An exact copy or Type-I clone of vulnerable code is also vulnerable and is straightforward to detect by simple matching. Near-miss or Type-III clones, for example, a buffer overflow bug where one clone has an extra logging statement, may still contain the core vulnerability and thus need detection. Semantic or Type-IV clones are the most insidious because the code logic may be restructured through different control flows or library calls. However, the same security weakness often persists, particularly in missing input validation cases. Often, detecting such semantic vulnerability variants requires understanding the code's behaviour rather than surface syntax.

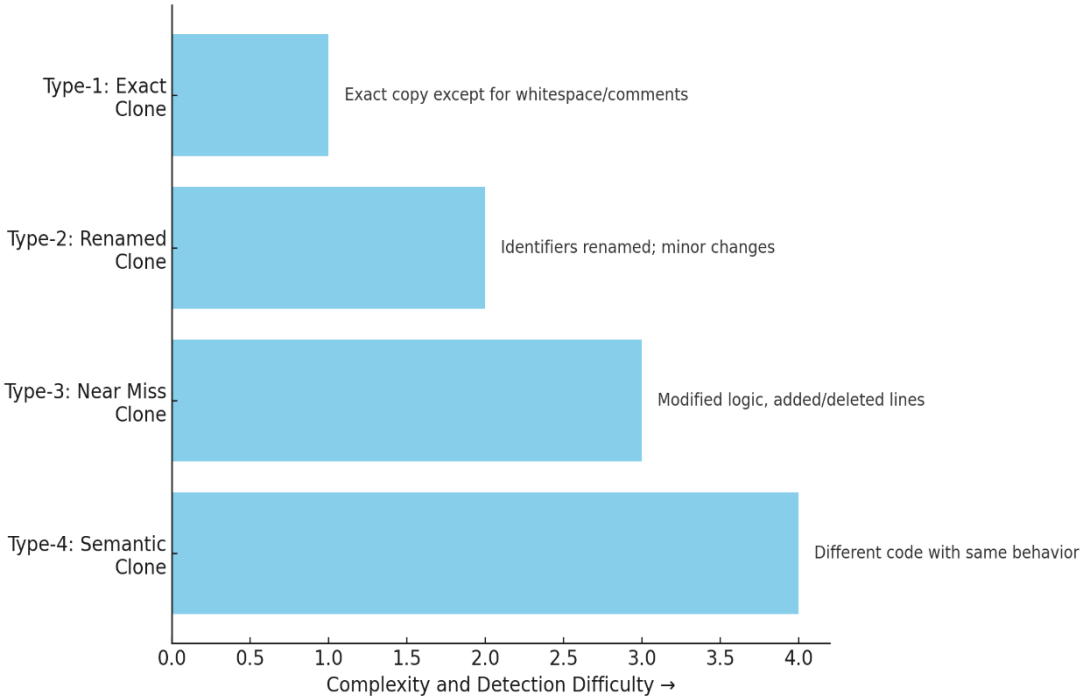

Fig 1: Clone types categorised by increasing code variation and detection difficulty

Vulnerabilities like buffer overreads, use-after-free and integer overflow depend on subtle code patterns and runtime behaviour. This behavioural complexity makes clone detection for security more challenging than general code clone detection for maintenance. An early empirical study by [6] observed that specific categories of code clones were more prone to security vulnerabilities than others, urging careful tracking of cloned code in security-sensitive projects. Furthermore, unpatched cloned vulnerabilities have led to actual incidents. For example, the widespread OpenSSL "Heartbleed" bug was patched in OpenSSL but lingered in numerous forks and similar implementations, creating latent risks [3]. These concerns motivate specialised techniques to scan code for vulnerable clones, essentially code instances identical or similar to known vulnerable code.

Before investigating detection approaches, A key challenge for vulnerable clone detectors is achieving an effective balance between precision and recall. High recall is needed to catch modified or semantic clones that are vulnerable to avoiding false negatives. However, aggressive matching can yield false positives if the code is similar but not vulnerable. Next, we review significant classes of clone detection techniques and how they attempt to strike this balance, especially in the face of the challenges outlined above.

2 Vulnerable Code Clone Detection Techniques: Researchers have adapted methods ranging from simple text matching to advanced deep learning to detect vulnerable code clones. We summarise each major category below, explaining the technique's typical strengths/limitations. We highlight how well each approach handles persistent challenges like semantic variance, obfuscation, and scale. This study identifies prominent tools or systems for each category.

2.1 Token-Based Techniques: Token-based clone detectors operate by transforming source code into a sequence of lexical tokens and then searching for identical or similar token sequences. Early examples include CP-Miner[7] and CCFinder [8], which tokenise source code elements like keywords, operators and delimiters and use efficient suffix trees or hashing algorithms to find vulnerable clones. Modern tools like NiCad [9] further enhance clone detection by normalisation of the source code through consistent renaming of all identifiers to detect near-miss clones. Token-based methods are generally fast and scalable because they reduce code to a linear token string. They can quickly leverage string matching or hashing to scan millions of lines.

However, the simplicity of token representation means these methods are limited to Type-I and Type-II clones in practice. If code is significantly changed (Type-III) or reorganised (Type-IV), token matching breaks down. An empirical analysis by Zhang et al. [10] confirms that token-based detectors can only detect Type I and II clones. Even modest control-flow changes easily evade them. For example, inserting an if check or reordering two statements in a copied code block will disrupt a naive token match. Token approaches are also brittle against obfuscation, i.e. transformations like splitting a line of code or adding dead code, which will alter the token sequence and hide the clone.

Despite these limits, token-based clone detection has been adapted for vulnerable code detection due to its speed. A notable case is VUDDY [11], a vulnerable code clone detector that uses the hashing of token sequences at the function level to find clones of known vulnerable functions. VUDDY sacrifices some recall, as it might miss modified clones in exchange for scalability. It indexed over one billion lines of code from thousands of software projects to rapidly locate vulnerable code clones. In evaluations, VUDDY showed extremely high precision ~99% but lower recall for transformed code. It reliably catches exact copies of vulnerable code but misses many Type-III cases, overall detecting only ~84% of known vulnerable instances in one study. In summary, token-based methods are efficient and work

well for nearly identical clones but struggle with semantic variations and obfuscations, making them insufficient alone for many vulnerable clone scenarios [2].

2.2 AST-Based Technique: Many tools use the code's abstract syntax tree (AST) structure to go beyond raw tokens. AST-based clone detectors parse source code into its syntax tree and then compare subtrees for similarity. The classic AST approach Deckard is demonstrated by Jiang et al. [12], which hashes vectors of AST features to detect similar codes with a tunable similarity threshold. By working on the parsed structure, AST-based methods become insensitive to formatting and even identifier renaming, as these changes do not affect the tree shape. They can thus catch Type-II clones easily and many near-miss Type-III clones as long as the overall structure is not radically altered. For example, if two functions differ only by a few statements or reordering of sibling statements, their AST representation will still be significantly similar, enabling detection.

AST-based techniques have shown higher recall on moderate code edits than token methods. In the BigCloneBench evaluation [4], AST-based and similar structural tools had a strong recall for Type-1, Type-2, and high-similarity Type-3 clones, significantly better than purely textual tools [13]. AST-based techniques are also more robust to certain obfuscations, such as changing variable names or reformatting code, because they do not change the AST. For example, swapping two independent statements yields an AST with similar subtrees that a robust algorithm can still detect. Despite these benefits, AST-based clone detectors have limitations regarding semantic clones and significant changes. The AST shape diverges if the `for`-loop clone gets rewritten as a `while`-loop. AST structures are sensitive to significant changes in the control flow or data flow [2] Therefore, AST-based detectors, like token-based ones, will struggle with Type-IV clones that are not structurally similar. They also can be computationally heavier than token approaches, as comparing large ASTs or doing subtree hashing across a big codebase can be expensive. Deckard [12] mitigated computational complexity via locality-sensitive hashing to avoid comparing all pairs. Another challenge is handling C/C++ idioms. In particular, C++ has very large ASTs due to templates and macros, making naive tree similarity cumbersome. Researchers have proposed various AST abstractions, such as pruning unimportant subtrees, generalising literal values and others to improve robustness. Still, semantic-equivalent but syntactically different code will often evade AST clone detectors. For vulnerable code clone detection, if a programmer modifies vulnerable code through refactoring, AST-based matching of the new code against the old vulnerable pattern may fail.

In practice, AST-based clone detection is often used with other signals. For instance, tools might filter candidate clone pairs by textual similarity and then apply an AST comparison to verify a match, thereby balancing speed and accuracy. Overall, AST-based methods represent a middle ground. They are more robust than pure tokens-based techniques as they handle near-misses and renaming but are not powerful enough to capture semantics fully. They form the backbone of many general-purpose clone detectors and contribute to vulnerability scanners. However, on their own, they cannot reliably catch the more complex cases of vulnerable clones that involve logic-level transformations.

2.3 Program Graph-Based Techniques: Researchers have looked to richer code representations, such as program dependency graphs, to detect semantic clones. Graph-based clone detection uses graphs that capture program semantics, such as the program's control flow graph (CFG), data flow, and program dependence graph (PDG). The intuition is that two code

fragments with the same functionality will have similar control and data flow, even if their syntax differs. Early work by Komondoor and Horwitz [14] and Krinke [15] explored using PDGs to find clones, although scalability was a concern. In the context of software security, Yamaguchi et al. [16] introduced the code property graph (CPG), which merges AST, CFG, and PDG into a joint graph representation to facilitate vulnerability discovery. Their system could find semantically similar suspicious code by traversing these graphs, essentially performing a graph pattern that matches known vulnerability patterns.

Graph-based clone detection can, in principle, catch clones that other methods miss. For example, suppose one version of a function uses a sequence of low-level C library calls, and another uses an equivalent higher-level API. In that case, their ASTs differ greatly, but a PDG might reveal that both have the same data dependencies and control conditions, indicating a semantic clone. Graph representations are more resilient to control or data flow changes than AST/text. For example, converting a `while` loop to an equivalent `for` loop changes the AST structure but largely preserves the control-flow graph, so a CFG-based detector can still recognise the similarity [2]. Despite their strengths, graph-based clone detectors historically suffered from practical limitations, most notably graph isomorphism. Checking if two subgraphs match is computationally expensive. Earlier PDG clone tools that attempted exhaustive graph matching proved time-consuming and often missed clones unless graphs matched exactly [17]. Recent research has attacked this with heuristics and graph abstractions. For instance, Vulnerable Code Graphs (VGraph [3]) introduced a graph-based code representation, "triplets", that splits a code fragment's graph into three parts, vulnerable slice, patch slice and context, to allow more tolerant matching. By breaking the graph comparison into smaller pieces and focusing on key relationship patterns, VGraph can handle modified clones more robustly. In evaluations, VGraph detected significantly more highly modified vulnerable clones than earlier methods. It achieved a recall of 96% and the highest F1-score, 97%, on a dataset of known vulnerable code, outperforming both VUDDY[11] and ReDeBug[18] by catching many missed clones. VGraph identified over twice as many Type-III and Type-IV modified clones as VUDDY[11] and over 100 more than ReDeBug[18] in a controlled test. These results underscore the strength of incorporating semantic graph relations. The trade-off with graph-based approaches is complexity and scalability. Constructing full PDGs or CPGs for large codebases of millions of lines and comparing subgraphs is computationally expensive. VGraph limited its scope by mining only functions related to known CVEs rather than searching all code for all vulnerable code clones. General-purpose PDG clone detection on arbitrary big code still requires significant computational resources or clever indexing. Some hybrid approaches create graph fingerprints, for example, hashing specific graph structures to enable faster lookup at the cost of some precision [19]. Another challenge is precision, and if the graph-matching process is too tolerant, it might incorrectly label functionally different codes as clones. Graph methods must carefully select which aspects of the graph are compared to avoid false positives. For vulnerable clone detection, typically, one aspect of a known vulnerable pattern is its signature in the graph of a target program [3], which helps with precision.

In summary, graph-based techniques represent the state-of-the-art for semantic clone detection. They tackle the semantic complexity challenge directly by focusing on code behaviour flows rather than surface syntax. When applied to security, as with Yamaguchi's CPG [16] or VGraph [3], they demonstrate the ability to find vulnerable code clone variants that other methods miss, for example, detecting a vulnerability that was silently patched in one project but remains in a modified form in another project. The cost is higher computational complexity and

implementation complexity. Graph-based clone detection is often targeted for specific known vulnerabilities or combined with simpler methods due to scalability constraints. It remains an active research area to make these methods more efficient and broadly usable in large-scale vulnerable code scanning.

2.4 Hybrid and Multi-Modal Techniques

Recognising that no single technique is sufficient for all clone types, many modern solutions use hybrid approaches, combining two or more methods. Hybrid clone detection might integrate textual, lexical, structural, and semantic techniques to improve overall recall and precision [20]. For example, a tool could first use a textual or token-based filter to quickly find candidate clones in a huge codebase and then apply a PDG-based check on those candidates to verify semantic similarity. Thus, it scales to big code while still catching semantic clones. SourcererCC [21]uses a similar pipeline, indexes token subsequence to candidate-match clones, and then computes an AST-based similarity on the candidates. The result is fast detection of even near-miss clones in big code repositories that would be infeasible with pure PDG matching.

ReDeBug [17] is a good example of a hybrid approach in the vulnerable code context. ReDeBug's goal was to find unpatched code clones where a patch had been applied in one part of an OS codebase but not in another instance of the clone. ReDeBug works by analysing the diffs of known patches, comparing the code before and after the fix and then scanning the codebase for the vulnerable code sequence removed or modified by the patch. It leverages token-based textual sequence matching but is constrained to the localised vulnerability context of a patch that integrates both text and semantic context. This approach allows it to catch cases where the vulnerable code might have slight edits, as long as these edits are outside the main patched lines. Indeed, ReDeBug was shown to find many vulnerable code clones that VUDDY missed by being more flexible in matching around the patched region. In one evaluation on Linux and other code, ReDeBug achieved ~92% recall of known vulnerable clones, higher than the strictly hash-based VUDDY, though with a slight precision trade-off. This illustrates how combining the patch knowledge, the semantics of what was fixed, with sequence matching yields a more effective detector for near-miss vulnerable clones. Similarly, ReDeBug's approach can be seen as integrating human knowledge and the patch diff with automated vulnerable code clone search to create a more effective strategy.

Another category of hybrid approaches merges static analysis with learning or pattern mining. For instance, VulPecker [22] takes a set of known vulnerable code patterns and, for each vulnerability type, chooses a suitable detection algorithm such as text-based, AST-based, or other that best finds that pattern. It is an adaptive and hybrid approach, where different vulnerability clone signatures are detected using different techniques. This technique worked well for the specific vulnerabilities tested as it could precisely find known vulnerabilities in C/C++ code using tailored clone criteria for each CWE. However, it requires manual mapping of vulnerabilities to the detection method, which does not scale easily to new vulnerability types. Researchers have proposed other hybrid approaches to overcome this limitation, such as symbolic execution with clone search. This hybrid approach first identifies syntactically similar clones. It then symbolically verifies if they exhibit the same vulnerable condition, thereby improving accuracy, as suggested by Liu et al. [23] in their research.

Key advantages of hybrid approaches are improved recall by catching clones one method would miss and precision by cross-checking using a second method. For example, a hybrid detector might only declare a clone if both a token similarity threshold and an AST structural similarity are met. It reduces false alarms. Alternatively, it might accept a clone if the token or

semantics match triggers. Thus, it improves recall. Empirically, hybrid detectors often outperform any single-technique detector. A recent survey notes that integrating textual, token, AST, and PDG features is a common strategy to enhance clone detection [24]

On the downside, hybrid methods can inherit the complexities of all components – making them harder to implement and sometimes slower. They also may require careful tuning of how the different signals are weighted or sequenced. Nonetheless, a hybrid strategy is arguably necessary for vulnerable code clone detection. One might use cheap hashing to scan enormous code collections for any candidates, then apply a deeper semantic analysis to confirm truly vulnerable clones. This approach is used by some industry tools that detect reused vulnerable OSS components, e.g., rough fingerprint matching to find possible copied code and then checking if the exact vulnerable lines are present [19].

In summary, hybrid approaches combine the strengths of multiple techniques. They are particularly promising for the multifaceted problem of vulnerable clone detection, where one needs lexical similarity to leverage known vulnerable code signatures and semantic analysis to catch evolved variants. The success of tools like ReDeBug and VulPecker demonstrates that thoughtfully blending techniques and incorporating external knowledge like patches or CWE-specific patterns yields more robust detection than any single approach.

2.5 Machine Learning Techniques (Pre-Deep Learning)

Machine learning has long been applied in code analysis, and before the deep learning era, researchers explored classic ML algorithms for clone detection and vulnerability detection. These traditional ML-based approaches typically involve extracting features from code and training a classifier or clustering algorithm to identify similar code or predict vulnerable code. Unlike deep learning, these methods typically use manually engineered features such as code metrics, code features and rely on simpler models like decision trees, SVMs

One example of clone detection is using software metrics or intermediate representations as features. For instance, researchers characterise each function by metrics such as the number of calls, number of branches, and types of operations forming a feature vector. Then a clustering algorithm groups the function with similar vectors to potential clones. An approach by Shar et al. [25] combined static analysis features like data flow patterns with ML using Naive Bayes to predict vulnerable code components in web applications by effectively learning what code metrics correlate with known vulnerabilities. Du et al. [26] proposed Leopard, identifying vulnerable C/C++ code by mining program metrics and smells such as complexity, nested loops, use of unsafe functions and using a classifier to flag functions likely to contain vulnerabilities. These can be seen as ML-assisted vulnerability detectors that might catch clones implicitly, as two functions that share vulnerable metrics might both be flagged.

In VulPecker, li et al. [22] automated feature selection. It tried multiple similarity algorithms for each vulnerability type and effectively *learned* which algorithm worked, a kind of meta-learning approach. Regarding persistent challenges, traditional ML-based approaches struggled with semantic generalisation and obfuscation just as non-ML ones did. They were only as good as their features. For example, the model will not detect that similarity if the features do not capture a particular code transformation.
Additionally, without massive training data, these models risk overfitting to patterns seen in training, for example, flagging specific API usage seen in known vulnerable samples but missing other ways that trigger the same vulnerability. Many earlier ML works also treated

vulnerability detection as a simple binary classification, vulnerable or not. However, this approach does not directly solve the problem of vulnerable code clone detection, which is more about finding correspondences between code snippets.
Nonetheless, this line of work set the stage for the deep learning-based methods that followed by demonstrating that automated learning from code is feasible. The features used, such as AST paths and opcode sequences, influenced the design of later neural network models. Some challenges were identified early on, such as the need for good datasets and generalisation. For example, Leopard [26] highlighted that the ML model's performance would be limited without a robust dataset of vulnerable code. This observation foreshadowed a key issue for all learning-based methods, which we will discuss in the dataset section.

Table 1: Summary of Vulnerable Code Clone Detection Techniques and Security-Specific Tools

Technique	Security-Specific Tools	Clone Types Handled	Strengths	Limitations
Token-based	VUDDY [11]	Type-I, limited Type-II	Fast, scalable, high precision	Fails on obfuscation, poor at Type-III/IV clones
Graph-based	VGraph[3], CCGraph [17], CPG [16]	Type-I to Type-IV (partial)	Captures semantic similarity, obfuscation-resilient	High computational cost, scalability challenges
Hybrid	ReDeBug [18], VulPecker[22], MOVERY [19]	Type-I to Type-IV (partial)	Balanced recall and precision	Integration complexity, tuning required
Machine Learning	Leopard [26], VulPecker [22]	Type-I, Type-II, limited Type-III	Learns from code metrics and patterns	Needs good features, limited semantic understanding
Deep Learning	VulDeePecker [27], Devign [28], GraphCodeBERT[29]	Type-I to Type-IV (partial)	Strong semantic representation, automatic feature extraction	Data-hungry, cross-project generalisation issues

2.6 Deep Learning-Based Techniques

In recent years, Deep Learning (DL) has transformed code analysis. Specifically, it has improved vulnerable code clone detection, significantly impacting system security and reliability. Software vulnerabilities can impact the security and reliability of a system, so it is important to detect them as soon as possible. Code clones duplicate snippets of code and often inherit security vulnerabilities. As code clones propagate across multiple software systems, the impact of that vulnerability also propagates and complicates vulnerability management and remediation. Deep learning methods can include convolutional neural networks (CNNs), recurrent neural networks (RNNs), or transformer-based models, all of which have been used to detect semantically similar code snippets that standard text-based or syntactic methods would miss. In vulnerability detection, deep learning models are designed to detect vulnerable

code patterns, similar to clone detection, because we treat the known vulnerable patterns as clones. VulDeePecker [27] is a notable system that uses a bi-directional LSTM to learn embeddings of code slices, specifically program slices around vulnerable points and then classify those slices as vulnerable or not. The model implicitly learns the vulnerable code pattern by training on many examples of vulnerable vs safe code, enabling it to flag code with similar patterns even if syntactically different. Similarly, Devign [28] applied a GNN to predict vulnerabilities in C/C++ functions, learning attention weights on graph nodes corresponding to suspicious operations like use-after-free patterns. These DL models can detect vulnerabilities that are semantic clones of known ones. Well-trained neural networks can infer the overall behaviour, such as a missing boundary check in a buffer overflow, even when the variable names and exact conditions differ. Moreover, these models leverage large-scale datasets, extracting intricate semantic features from source code, thus improving detection accuracy and reducing false positives compared to conventional techniques.
However, new Challenges emerged with deep learning approaches.

- **Training Data Requirements:** Deep learning models require large, representative datasets of clones or vulnerabilities to train effectively. Otherwise, they can overfit or not generalise. VulDeePecker [27] states that a robust dataset is necessary for the ML algorithm to learn to identify vulnerable code. Until recently, such datasets were limited. As a result, many early DL models were trained on imperfect data, such as BigCloneBench[4] for clones or Draper [30] for vulnerabilities. They might not handle real-world code as well, despite their reported performance. Semantic variations or labelling noise within datasets significantly degrade model effectiveness.
- **Generalisation and Project Bias:** Chen et al. [31] that deep learning models can perform dramatically worse on code from projects not included in their training data. In an experiment, models trained on vulnerabilities from open-source projects saw their F1 score drop from ~49% on seen projects to only ~9.4% on unseen projects [31]. This scenario indicates that models may learn distinct patterns, such as coding style or API usage specific to certain codebases, rather than truly learning the general pattern of the vulnerability. Such cross-project generalisation failure is a serious issue for deployment. In practice, we want to find vulnerabilities in new code, not just in the projects from which we had examples. This significant drop in performance when models are evaluated on unseen projects highlights a core limitation of current deep learning-based approaches. Figure 2 illustrates this cross-project generalisation gap for several well-known models, showing a consistent decline in F1 scores across all of them. The model needs to handle a form of cross-language or platform generalisation, even if the language is the same, the context differs per project.
- **Interpretability and Manual Validation:** Deep learning models output predictions without explicit explanations unless additional techniques are used. For security use, this is tricky, and developers are cautious to trust a black box saying this code is vulnerable without rationale. Traditional clone detectors can at least show the matching code snippet from a known CVE as evidence. With DL, one often has just a probability. While this does not directly affect detection performance, it affects the workflow and adoption of such tools in practice. Some recent research tries to combine neural approaches with logic rules to get the best of both. For example, neural models propose candidates and a symbolic engine verifies the vulnerability.

Deep learning methods have significantly advanced vulnerability detection and vulnerable code clone detection by effectively capturing complex semantic features in code. Recent transformer-based models, such as CodeBERT and GraphCodeBERT [29], have produced robust embeddings, resulting in state-of-the-art performance on established benchmarks.

Similarly, large language models (LLMs), including GPT-based architectures, show promise in identifying vulnerabilities across different programming languages. However, recent studies [32] highlight ongoing limitations, such as GPT-3.5's difficulty in accurately detecting cross-lingual code clones, emphasising the continued challenge of creating unified code representations.

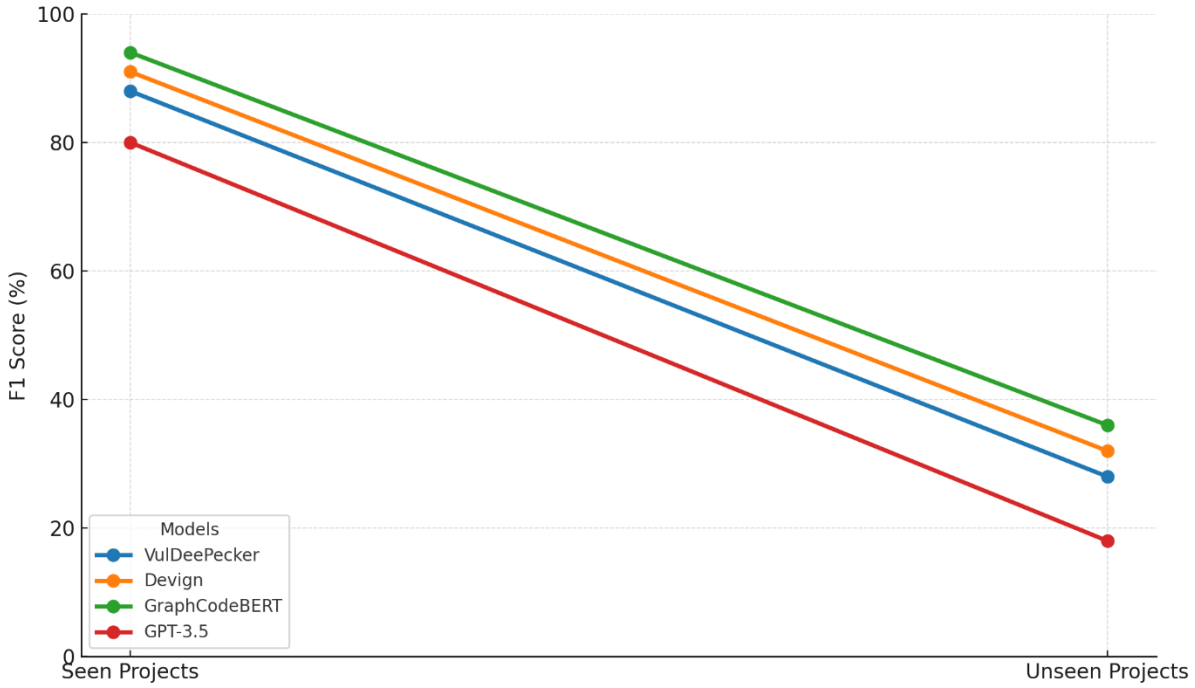

Figure 2: Cross-project generalisation gap in F1 scores across deep learning models

Despite these successes, deep learning techniques still face critical issues related to data dependence, generalisation across diverse projects, and robustness to intentional obfuscations. The research community is actively developing more representative datasets like DiverseVul[31] incorporating project-agnostic features, and combining deep learning with traditional static analysis methods to address these challenges. Consequently, deep learning approaches, though highly promising, should be carefully integrated with curated training data and hybrid techniques to fully address the complexities inherent in vulnerable code clone detection tasks. A comparative summary of all major vulnerable code clone detection techniques discussed thus far, along with representative tools, clone types handled, and key trade-offs, is presented in Table 1 which highlights that each technique offers distinct strengths but also inherits critical limitations. These limitations, particularly in terms of scalability, semantic complexity, and dataset dependency are explored in detail in the next section.

3 Persistent Challenges in Vulnerable Clone Detection

Bringing together the observations from techniques and datasets, we now highlight the persistent, unresolved challenges across methods. These complex problems make vulnerable code clone detection an ongoing research challenge.

3.1 Semantic Complexity of Clones: Semantic clone detection, which identifies code that does the same thing differently, remains an elusive goal. Despite progress, Type-IV or semantic clones are difficult for all approaches. As we have seen, traditional detectors often fail in Type-IV cases, and even advanced ones like graph-based methods might catch only certain kinds,

such as the same data-flow pattern, but not others with different algorithms altogether. In a vulnerability context, this is critical because a clever developer might modify a known bug by reimplementing functionality differently, or two programs might use different logic to achieve a task, but both miss the same security check.

One illustrative example is that two libraries might parse an image format differently, such as one using a state machine and another using recursion. However, both forget the bounds check, leading to an overflow. These would be semantic clones of a vulnerability. Detecting such cases pushes the limits of static analysis and AI, essentially venturing into program equivalence, a general undecidable problem.

Deep learning offers some hope hereby generalising patterns, but it has not fully solved it. A model might learn standard features of buffer overflows, such as using the `memcpy` function or `pointer` arithmetic in a certain way and catch semantically similar bugs. However, if two code fragments implement a check differently, even a neural network may not correlate them unless it has seen similar pairs in training.

The semantic gap also ties into understanding the intent of code. Two pieces of code may look similar but do different things, or they might look completely different but do the same thing. For example, one sorts a list, and another searches a list, but they might have a similar structure or vice versa. Distinguishing that purely via automated means is hard. We care about intent insofar as the security property is concerned in vulnerable clone detection. In other words, regardless of how the code is implemented, it either performs or fails to perform the critical steps needed for safety. If two codes omit a needed input validation, they are semantically similar in the sense of vulnerability, even if the rest of their logic differs.

Current research directions to address semantic complexity include:

- **Combining static analysis and specification** with clone detection to focus on security-relevant semantics. For example, taint analysis can identify if two functions allow unchecked data to reach a sink.
- **Semantic-preserving transformations,** such as those applied by CLONEGEN's [2] operators during evaluation, can be integrated into the training process of machine learning models to enhance their robustness. Incorporating these transformations allows the models to become invariant to structural changes that do not alter the underlying functionality of code.
- **Leveraging functional test outputs**: One could execute code when feasible to see if two functions produce the same outputs for various inputs to detect semantic equivalence. While this shifts the task towards software testing rather than traditional static clone detection, it might augment static analysis for small code snippets.
- **Language-agnostic representations**: Intermediate languages such as LLVM IR or semantics vectors can help reveal Type-IV clones. Type-IV clones might become more apparent if the code is compiled to an IR. However, if logic differs greatly, even IR will differ.

Semantic complexity is likely to persist as a challenge because it is rooted in the fundamental hardness of understanding program behaviour. The progress in this area will determine how effectively deep vulnerable clones we can catch. Right now, many tools tend to adopt a cautious approach and focus on Type-II/III clones to avoid the intractable Type-IV search, meaning truly crafty variant vulnerabilities could slip through.

3.2 Obfuscation and Code Transformation

Obfuscation refers to transformations of code intended to conceal its similarity to other code while preserving functionality. In malware and exploit development, attackers often obfuscate known vulnerable code when reusing it, specifically to evade signature-based detectors. For example, a published exploit might be cloned with variables renamed, the order of checks changed, or sprinkled with irrelevant operations to throw off simple matching. Therefore, vulnerability scanners that rely on pattern matching often contend with obfuscated clones.
Many of the techniques we discussed are brittle against obfuscation:

- Token-based and AST-based detectors were shown to be easily evaded by transformations such as statement reordering, control flow alternation, or even specific simple opcode changes [2].
- The CLONEGEN experiments systematically demonstrated that a series of small changes, such as changing a `for`-loop to a `while`-loop or unrolling a loop, can cause substantial drops in detection by ML and traditional tools [2].
- Even graph-based detectors can be deceived by transformations that alter the graph enough. For instance, adding dummy branches or differently reusing variables might change the dependency graph to hide the core similarity.

Obfuscation vs Semantic clone: Obfuscation is essentially a subset of semantic clones where the changes are deliberately designed to throw off detection. It is an adversarial scenario. Therefore, the challenge overlaps with semantic complexity but with the twist that the clone difference is not naturally occurring but is engineered to be hard for known detectors. This adversarial dynamic leads to an ongoing cycle where detection techniques adapt to specific forms of obfuscation, and adversaries often respond with increasingly complex methods.

Many real-world vulnerable clones arise from copy-paste or minor edits that are not heavily obfuscated. However, in exploits, attackers often obfuscate code deliberately. Even benign refactoring in open-source projects can unintentionally obscure vulnerabilities. Security experts must assume that known vulnerabilities may appear in modified form. Robust detection must account for such variations. For instance, VGraph[3] achieved 77% recall on modified clones, compared to 34% for VUDDY [11]. These results highlight the importance of developing detection methods resilient to obfuscation-like changes.

In summary, obfuscation continues to be a challenge that requires detectors to be flexible in matching. The best current defence combines normalisation, semantic analysis, and learning from multiple variants so the detector is not tied to one appearance. The arms race aspect means we must continually evaluate detectors under normal and adversarial conditions. Zhang et al. [2] exemplify this approach by evaluating detectors using semantic transformations to simulate adversarial conditions. Many vulnerable clone detectors did not initially consider this, so robustness is now an active area of improvement.

3.3 Scalability to Large Codebases

Scalability, or handling massive code corpora, is a pragmatic challenge. Industrial use cases for vulnerable clone detection involve scanning millions of lines of code, such as all of a Linux distribution or all GitHub projects, to find instances of a known vulnerability. A too slow or memory-intensive technique cannot be applied at this scale.

We observed that Simpler methods, such as token-based and textual, are very scalable. Tools like CCFinder [8]and NiCad [9] can process millions of lines efficiently, and VUDDY has indexed billions of tokens by reducing each function to a hash. Scalability was a key design goal of VUDDY, trading off thoroughness. Indeed, VUDDY's authors demonstrated scanning thousands of firmware images for vulnerable code in hours, which is not feasible with slower methods.

AST-based methods are moderate: parsing is usually linear in code size, but comparing ASTs for clones can be costly if done naively $O(n^2)$ comparisons for n functions. Deckard [12]mitigated this by hashing and dimensionality reduction; SourcererCC [21] using an index. Modern AST-based clone finders can handle large datasets by these means, but the extreme scale (hundreds of millions of LOC) might still be challenging without a big cluster.

Graph-based methods traditionally scale poorly. Constructing PDGs for extensive projects is heavy, and the clone query, which involves subgraph isomorphism, can blow up combinatorially. VGraph's approach to limit analysis to relevant parts, using CVE references to fetch only suspect files, is one way to sidestep the scale issue. Another approach is incremental analysis: a recent paper proposes incremental PDG-based clone detection to update clone info as code evolves rather than re-run from scratch [33]. This method is practical in continuous integration scenarios but does not fully solve scanning arbitrary large code dumps.

Machine learning/deep learning methods have a different scalability concern: training and inference. Training a deep model on millions of code samples is time-consuming, though it is usually one-time. Inference, which involves embedding code and comparing it, can be made fast if the model is efficient and one uses an approximate nearest neighbour search in the embedding space. A key challenge arises when the model, such as Transformers and the input code, are large, leading to slow and memory-intensive encoding. A common optimisation is to limit input size, often by processing code at the function level or even smaller code fragments.

3.4 Dataset Limitations and Evaluation Blind Spots

Ground Truth Incompleteness: To evaluate a vulnerable clone detector, one needs a set of vulnerable code instances and knowledge of all the clones in the target code. Such ground truth is brutal to assemble in real projects as many clones go undetected even by experts. BigCloneBench provided ground truth for general clones but did not specifically target vulnerable ones. Evaluations like VGraph's had to generate ground truth by mining CVE mentions and then manually confirming clones [3]. Likely, their ground truth is still incomplete as they considered certain functions as the base and looked for clones of those. Any clone not linked to a known CVE would be absent. As a result, a detector's recall might be measured against an incomplete set of actual clones. To better understand the source of these limitations, table 2 provides a comparative summary of commonly used datasets in vulnerable code clone detection. It outlines their coverage, strengths, and known weaknesses, providing a basis for the critical issues discussed in this section.

Label Noise: As discussed, datasets like VDISC [30] or Big-Vul [34] can have noisy labels. A detector might find a real vulnerability, but if the dataset label says not vulnerable due to the wrong label, it counts as a false positive or vice versa. Such mislabelling can lead to misleading evaluations of a tool's precision and recall. To address the issue of mislabelled vulnerabilities, ongoing research efforts are focused on cleaning and improving the quality of these datasets [35].

Benchmark representativeness: Tools often get optimised to perform well on available benchmarks. For example, a clone detector might overfit the patterns in BigCloneBench[4] or

Juliet's synthetic styles[36]. If a new benchmark with different characteristics is used, performance can drop. We saw this with deep learning models that did great on randomly split data but poorly on cross-project splits [31]. It suggests our benchmarks might not have been measuring the right thing initially (overestimating generalisation). The research community is now aware of this and is shifting to more complex evaluation setups.

Table 2: Benchmark Datasets for Vulnerable Code Clone Detection

Dataset	Type	Strengths	Limitations
BigCloneBench[4]	General Clone Detection	Large, clone-type labeled, diverse syntactic clones	Not vulnerability-focused, limited semantic clone coverage
Juliet[36]	Synthetic Vulnerabilities	CWE-based, balanced vulnerable/non-vulnerable pairs	Unrealistic patterns, lacks real-world complexity
VDISC/Draper [30]	Real Vulnerabilities	Real CVEs, large-scale function corpus	Label noise, weak semantic clone labeling
Big-Vul [34]	Real Vulnerabilities	Real-world vulnerable commits, linked CVEs	Mislabelled or incomplete patches, lacks non-vulnerable contrast
DiverseVul [31]	Mixed, Real-world + CVEs	Cross-project diversity, evolving vulnerability types	Still growing, not widely adopted yet

Lack of real-world diverse scenarios in benchmarks: Juliet and similar synthetic sets cover many CWE types, but they do not cover interactions of vulnerabilities or complex authentic codebases. Real code often has multiple issues interacting, such as a buffer overflow that only occurs after a specific loop condition, which a small test case might not capture. Clone detectors might need to see these in context. Furthermore, things like vulnerable code in large files, with lots of noise, are hard to emulate in small test cases.

Evolving nature of vulnerabilities: New types of vulnerabilities or new coding patterns keep arising, such as the surge in use-after-free issues with modern C++ patterns. Datasets tend to be backward-looking and are based on known CVEs. As a result, a tool trained on these datasets could be great for historical patterns but blind to new ones. This limitation is not precisely a dataset bug but a reality that evaluation must keep up with the evolving landscape.

In summary, reliably evaluating vulnerable clone detection is difficult due to data issues. Improvements in datasets like DiverseVul's [31] larger size and mix of projects will help, but researchers must be vigilant about validation methodology. Cross-validation across projects, basic consistency checks for label noise, and multiple datasets, such as synthetic and real, are recommended to judge a tool's actual performance. Without careful evaluation, a tool might seem to perform excellently but fail in a real deployment, ultimately hindering progress and adoption.

4 Conclusion

Vulnerable code clone detection has evolved significantly, moving beyond foundational surveys into a domain rich with diverse complementary techniques. We have seen that different detection techniques complement each other: from token-based hashing that scalable pinpoints

exact matches to AST and graph analyses that generalise to near-miss and semantic clones to machine learning models that promise even broader pattern recognition. Each approach addresses some facets of the problem while leaving others unsolved. Despite this progress, the field remains an evolving adversarial challenge that requires models to capture semantics, withstand obfuscation, and operate at scale.

Continued research is essential to overcome practical limitations such as generalising unseen codebases, maintaining robustness under adversarial transformations, and adapting to multilingual and cross-platform software ecosystems. The quality of the datasets and evaluation protocols underpinning these models are just as important. Efforts like DiverseVul and project-specific splits signal a shift toward more realistic and reliable benchmarks.

Ultimately, advancing this field will require a holistic and interdisciplinary approach combining software engineering, program analysis, and AI to build detection systems that are accurate, resilient, scalable, and deployable in real-world environments.

5 Future Scope: Future research will likely explore hybridising deep learning with traditional static analysis. One promising direction is using deep learning to suggest potentially vulnerable clones and verify them with symbolic execution or model checking to ensure the vulnerability condition holds. This approach could dramatically reduce false positives while maintaining high recall, essentially combining the strengths of both worlds. Another direction is developer assistance**:** tools integrated into IDEs that warn developers if the code they just wrote is similar to a known vulnerable snippet, perhaps using local vector search against a vulnerability database. This preventive application would be the next step beyond just post-factum scanning.

[1] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” *Inf Softw Technol*, vol. 55, no. 7, pp. 1165–1199, Jul. 2013, doi: 10.1016/j.infsof.2013.01.008.

[2] W. Zhang, S. Guo, H. Zhang, Y. Sui, Y. Xue, and Y. Xu, “Challenging Machine Learning-Based Clone Detectors via Semantic-Preserving Code Transformations,” *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3052–3070, May 2023, doi: 10.1109/TSE.2023.3240118.

[3] B. Bowman and H. H. Huang, “VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets,” in *Proceedings - 5th IEEE European Symposium on Security and Privacy, Euro S and P 2020*, Institute of Electrical and Electronics Engineers Inc., Sep. 2020, pp. 53–69. doi: 10.1109/EuroSP48549.2020.00012.

[4] “BigCloneBench- Benchmark Dataset for Clone detection.” Accessed: Apr. 29, 2023. [Online]. Available: https://github.com/clonebench/BigCloneBench

[5] B. Dolan-Gavitt *et al.*, “LAVA: Large-Scale Automated Vulnerability Addition,” *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pp. 110–121, Aug. 2016, doi: 10.1109/SP.2016.15.

[6] M. R. Islam and M. F. Zibran, “A comparative study on vulnerabilities in categories of clones and non-cloned code,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016*, Institute of Electrical and Electronics Engineers Inc., May 2016, pp. 8–14. doi: 10.1109/SANER.2016.90.

[7] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, Mar. 2006, doi: 10.1109/TSE.2006.28.

[8] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, Jul. 2002, doi: 10.1109/TSE.2002.1019480.

[9] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *2011 IEEE 19th International Conference on Program Comprehension*, IEEE, Jun. 2011, pp. 219–220. doi: 10.1109/ICPC.2011.26.

[10] H. Zhang and K. Sakurai, "A Survey of Software Clone Detection From Security Perspective," *IEEE Access*, vol. 9, pp. 48157–48173, 2021, doi: 10.1109/ACCESS.2021.3065872.

[11] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," in *Proceedings - IEEE Symposium on Security and Privacy*, Institute of Electrical and Electronics Engineers Inc., Jun. 2017, pp. 595–614. doi: 10.1109/SP.2017.62.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," *Proceedings - International Conference on Software Engineering*, pp. 96–105, 2007, doi: 10.1109/ICSE.2007.30.

[13] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with BigCloneBench," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Sep. 2015, pp. 131–140. doi: 10.1109/ICSM.2015.7332459.

[14] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2126 LNCS, pp. 40–56, 2001, doi: 10.1007/3-540-47764-0_3.

[15] J. Krinke, "Identifying similar code with program dependence graphs," *Reverse Engineering - Working Conference Proceedings*, pp. 301–309, 2001, doi: 10.1109/WCRE.2001.957835.

[16] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," *Proc IEEE Symp Secur Priv*, pp. 590–604, Nov. 2014, doi: 10.1109/SP.2014.44.

[17] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "CCGraph: a PDG-based code clone detector with approximate graph matching ACM Reference Format", doi: 10.1145/3324884.3416541.

[18] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire OS distributions," in *Proceedings - IEEE Symposium on Security and Privacy*, Institute of Electrical and Electronics Engineers Inc., 2012, pp. 48–62. doi: 10.1109/SP.2012.13.

[19] S. Woo, H. Hong, E. Choi, and H. Lee, "{MOVERY}: A precise approach for modified vulnerable code clone discovery from modified {Open-Source} software components," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3037–3053.

[20] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges," *Journal of Systems and Software*, vol. 204, p. 111796, Oct. 2023, doi: 10.1016/J.JSS.2023.111796.

[21] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proceedings - International Conference on Software Engineering*, IEEE Computer Society, May 2016, pp. 1157–1168. doi: 10.1145/2884781.2884877.

[22] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, New York, NY, USA: ACM, Dec. 2016, pp. 201–213. doi: 10.1145/2991079.2991102.

[23] Z. Liu, Q. Wei, and Y. Cao, "VFDETECT: A vulnerable code clone detection system based on vulnerability fingerprint," in *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)*, IEEE, Oct. 2017, pp. 548–553. doi: 10.1109/ITOEC.2017.8122356.

[24] H. Zhang and K. Sakurai, "A Survey of Software Clone Detection From Security Perspective," *IEEE Access*, vol. 9, pp. 48157–48173, 2021, doi: 10.1109/ACCESS.2021.3065872.

[25] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning," *IEEE Trans Dependable Secure Comput*, vol. 12, no. 6, pp. 688–707, Nov. 2015, doi: 10.1109/TDSC.2014.2373377.

[26] X. Du *et al.*, "LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment Through Program Metrics," *Proceedings - International Conference on Software Engineering*, vol. 2019-May, pp. 60–71, May 2019, doi: 10.1109/ICSE.2019.00024.

[27] Z. Li *et al.*, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018*, The Internet Society, 2018. doi: 10.14722/ndss.2018.23158.

[28] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," 2019. [Online]. Available: https://sites.google.com/view/devign

[29] D. Guo *et al.*, "GraphCodeBERT: Pre-training Code Representations with Data Flow," *ICLR 2021 - 9th International Conference on Learning Representations*, Sep. 2020, Accessed: Apr. 19, 2023. [Online]. Available: https://arxiv.org/abs/2009.08366v4

[30] L. Kim and R. Russell, "Draper VDISC Dataset - Vulnerability Detection in Source Code," Art. no. 12761885, 2018, doi: none.

[31] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection," *ACM International Conference Proceeding Series*, vol. 15, no. 23, pp. 654–668, Oct. 2023, doi: 10.1145/3607199.3607242.

[32] J. Mao, Z. Tang, and W. Rao, "Cross-Language Binary-Source Code Matching Based on Rust and Intermediate Representation," in *2023 IEEE 3rd International Conference on Computer Communication and Artificial Intelligence, CCAI 2023*, Institute of Electrical and Electronics Engineers Inc., 2023, pp. 195–199. doi: 10.1109/CCAI57533.2023.10201266.

[33] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto, "Incremental code clone detection: A PDG-based approach," *Proceedings - Working Conference on Reverse Engineering, WCRE*, pp. 3–12, 2011, doi: 10.1109/WCRE.2011.11.

[34] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries," *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, pp. 508–512, Jun. 2020, doi: 10.1145/3379597.3387501.

[35] R. Croft, M. A. Babar, and M. M. Kholoosi, "Data Quality for Software Vulnerability Datasets," *Proceedings - International Conference on Software Engineering*, pp. 121–133, 2023, doi: 10.1109/ICSE48619.2023.00022.

[36] "Juliet C/C++ 1.3 - NIST Software Assurance Reference Dataset." Accessed: Apr. 02, 2023. [Online]. Available: https://samate.nist.gov/SARD/test-suites/112