International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(11s) 2026 pp. 895-914
https://doi.org/10.70917/ijcisim-2026-3822
Article

AMBER: A Semantic Aware Detection of Code Vulnerabilities and Their Clones

Gurpreet Singh [1], Dhavleesh Rattan [2]

[1,2]Department of Computer Science and Engineering, Punjabi University, Patiala, Punjab, India
[1]gurpreet.1887@gmail.com | ORCID: 0009-0005-1383-8720,

Abstract: The identification of vulnerabilities in software and their clones is crucial for secure software development, but achieving high accuracy whilst maintaining a scalable detection system is difficult. We introduce Amber, a semantic-aware framework that utilizes semantic embeddings, structural program features, and similarity metrics to construct a multi-channel representation, which is fed to a convolutional neural network (CNN) for vulnerability detection, and then further employed in a FAISS-based semantic similarity search for finding vulnerable code segments that can be reused in other large codebases. In addition to real-world data from open-source repository data sets, Amber is assessed on 12,303 vulnerable functions and 21,057 non-vulnerable functions from the SARD vulnerability database. Regarding accuracy, recall, and F1-score, Amber demonstrated superior performance over several leading baselines, including VulCNN, VulDeePecker, SySeVR, and Devign. In addition, Amber enables post-classification vulnerable clone detection, providing a deep understanding of vulnerabilities through hidden and subtle vulnerable clones. This work provides a scalable and explainable framework for development and security practitioners, helping to advance the automation of vulnerability and their clone detection and improve software security practices overall.

Keywords: Software Vulnerability Detection, Vulnerable clone detection, Program Dependence Graph, Semantic Code Analysis, Multi-Chanel Neural Representation.

1. Introduction

The software is the layer that sits upon today's technology, allowing people and businesses to execute many tasks with greater speed and capability than ever before. In nearly every industry and sector, business process transformational efforts will have continuous software needs linked to a universal need for new software solutions, making that demand ever-increasing. Open-source software is pivotal in fulfilling these requirements and has experienced rapid growth in the past few years. For example, in November 2024, GitHub reported that the number of developers had reached 100 million, and the number of repositories had increased to 518 million . With the exponential increase in open-source software, the number of software vulnerabilities in these projects also rises. Software vulnerability is a flaw or weakness in the code that can be exploited to compromise software security, integrity and confidentiality. Software vulnerabilities originate from various factors, such as design flaws, poor and insecure coding practices, time constraints, and inadequately skilled developers. History has shown how hazardous a software code vulnerability can be. For example, a coding oversight led to the emergence of a security vulnerability named Heartbleed (CVE-2014-0160) in the OpenSSL library in 2012. However, it remained unnoticed until researchers discovered and publicly disclosed it in April 2014. Heartbleed was one of the incidents that exposed countless websites, servers, and applications to critical data leakage. This incident underscores how a latent bug can silently persist and propagate risk for years.

Code reuse is also a concern within software development that frequently results in code clones, which may be intentional or unintentional . Developers frequently use code cloning practices to accelerate the development process. Developers should closely monitor code cloning practices since they can be harmful by propagating vulnerabilities, particularly when cloned code is vulnerable . It results in vulnerable code clones, i.e., insecure code spread across different parts of a code base. Vulnerable code clones are a serious concern for stakeholders, such as software developers, security experts, organisations, open-source communities, governments and end users. Identifying such vulnerable clones is vital because different stakeholders are concerned about fixing these vulnerable code clones as

soon as possible without being compromised by hackers. Moreover, developers must ensure that a patch for one instance is propagated to all clones when a vulnerable code snippet is copied to multiple locations. Otherwise, attackers can target the unpatched code copies.

Traditional vulnerability detection involves analysing source code to find dangerous patterns or bugs that could be exploited. Vulnerability detection typically relies on expert-defined rules or features and is labour-intensive and prone to false negatives when novel vulnerabilities or obfuscated patterns are present. On the other hand, vulnerable code clone detection, or identifying cloned vulnerable code across multiple software projects, is a new challenge since vulnerabilities can spread throughout projects due to developers' frequent code reuse. Therefore, identifying and fixing them is essential. Integrating both approaches is essential for effective deployment, creating a more potent and effective security framework.

Over the past ten years, deep learning methods have become very effective for automating clone and vulnerability detection. Researchers have developed methods that allow neural models to automatically extract code features from large-scale code datasets without requiring experts to develop manual rules. Several studies have focused on treating code as a data sequence, like token sequences or AST paths for recurrent neural networks. In contrast, others treat code as graphs for graph neural networks or images for convolutional neural networks. These data-driven approaches have demonstrated exciting results for detecting vulnerable syntactic constructs . They can generalise well beyond handcrafted rules to enable systems to scale across larger codebases and expose more subtle patterns of vulnerability.

Deep learning approaches have also been investigated for the problem of vulnerable code clone detection. When models learn rich representations of code, they can track instances of vulnerable code that have been copied and potentially modified, even partially obscuring simple syntactic matching upon which traditional methods rely. Detecting and remediating these vulnerable code clones is important to avoid the silent propagation of common vulnerabilities across different software projects. By investigating such approaches, we would develop tools that can be applied to large codebases while also checking for potentially more subtle vulnerability code clones or patches of code that cover them.

We are introducing Amber, an enhanced framework based on the VulCNN model. The name Amber is derived from the naturally occurring fossilised resin that preserved biological specimens for millions of years, representing our ambition to capture and preserve semantic patterns of vulnerabilities across software systems. Amber enhances VulCNN by using an enriched representation that includes additional semantic, structural, and similarity features and a post-classification clone detection phase supported by FAISS-based semantic similarity retrieval. With these enhancements, Amber increases the precision of vulnerability detection and allows for the detection of vulnerable code clones, addressing a significant gap in the existing deep-leaning vulnerability detection. Amber is semantic-aware CNN based two-stage Vulnerable Function Detection and Code Clone Detection. The next sections will discuss the detailed architecture and implementation of Amber, detailing its use of FAISS for efficient nearest neighbour retrieval.

This study has made the following main contributions:

- **Enhanced Input Representation:** In order to provide more semantic and structural context for vulnerability and vulnerable clone detection, Amber adds three additional channels to the VulCNN's three-channel RGB input: function-level semantic embeddings, cyclomatic complexity, and top-*k* similarity.
- **Semantic-Aware Framework Design:** AMBER's framework design represents fine grained syntactic structure along with high level semantic information. This allows detection of vulnerabilities on finer scales while also being aware of higher-level context.
- **Clone Identification after Vulnerability Detection:** Amber includes a two-step process. Vulnerable functions are identified according to the vulnerability classification in Amber and retrieved by FAISS-based nearest-neighbour retrieval and matched by semantic similarity thresholding to find vulnerable code clones.
- **Scalability through FAISS Integration:** With a larger codebase, Amber can become truly scalable when it comes to supporting similarity searching of vulnerable embeddings with FAISS.
- **Comprehensive Evaluation:** Amber is empirically evaluated using a benchmark SARD [14] set of 12303 vulnerable and 21057 non-vulnerable functions. The proposed Amber method was found to be superior to an existing deep learning method in identifying vulnerabilities and identifying vulnerable clones.

2. Proposed Methodology: The Amber Framework

Amber's methodology consists of multiple stages that transform source code into a rich feature representation, then apply a CNN-based classification to detect vulnerabilities and perform clone detection for vulnerable instances. Figure 1 provides an overview of the Amber system architecture and workflow. In this section, we explain each stage and how they interconnect, and we present Amber's algorithmic flow. Amber's workflow can be broadly divided into the following six stages:

- **Code Pre-processing and Normalisation:** The source code is cleaned and normalised to establish canonical syntax, remove noise, and prepare the source code for further analysis.
- **Graph Construction and Feature Extraction:** Program dependence graphs (PDGs) are constructed to represent the relationships among syntactic, semantic, and structural aspects of each function and top-k similar vulnerable functions are retrieved through FAISS to build relationships and broaden the feature set.
- **Semantic code Embedding and Indexing:** Each function is represented as a semantic embedding, and an index of known vulnerable embeddings is constructed for fast similarity searches.
- **Feature Fusion and Representation Enhancement:** The features from the multiple feature channels (embeddings, graph-derived features, and similarity scores) are fused to establish an input.
- **Multi-Channel Representation Construction:** The features from the multiple feature channels, like embeddings, graph-derived features, and similarity scores, are fused to establish an input.
- **Vulnerability Detection:** A CNN is trained on the SARD dataset to classify each function in the input as vulnerable or non-vulnerable using fused input that could consist of multi-channel sources.
- **Vulnerable Code Clone Detection:** A function identified as vulnerable is then analysed to find its clone instances across the codebase by conducting an embedding comparison against known vulnerabilities.

2.1 Code Pre-processing and Normalisation: We cleaned and normalised the source code to ensure consistency in the source code. It started with removing white spaces, tabs, extra line feed characters, and comments in the source code. To further standardise the code, we converted all code characters to lowercase. Then, we replaced all variable names, function names, and parameter names with general names except some built-in keywords since some of these keywords are directly responsible for the occurrence of vulnerabilities in the code.

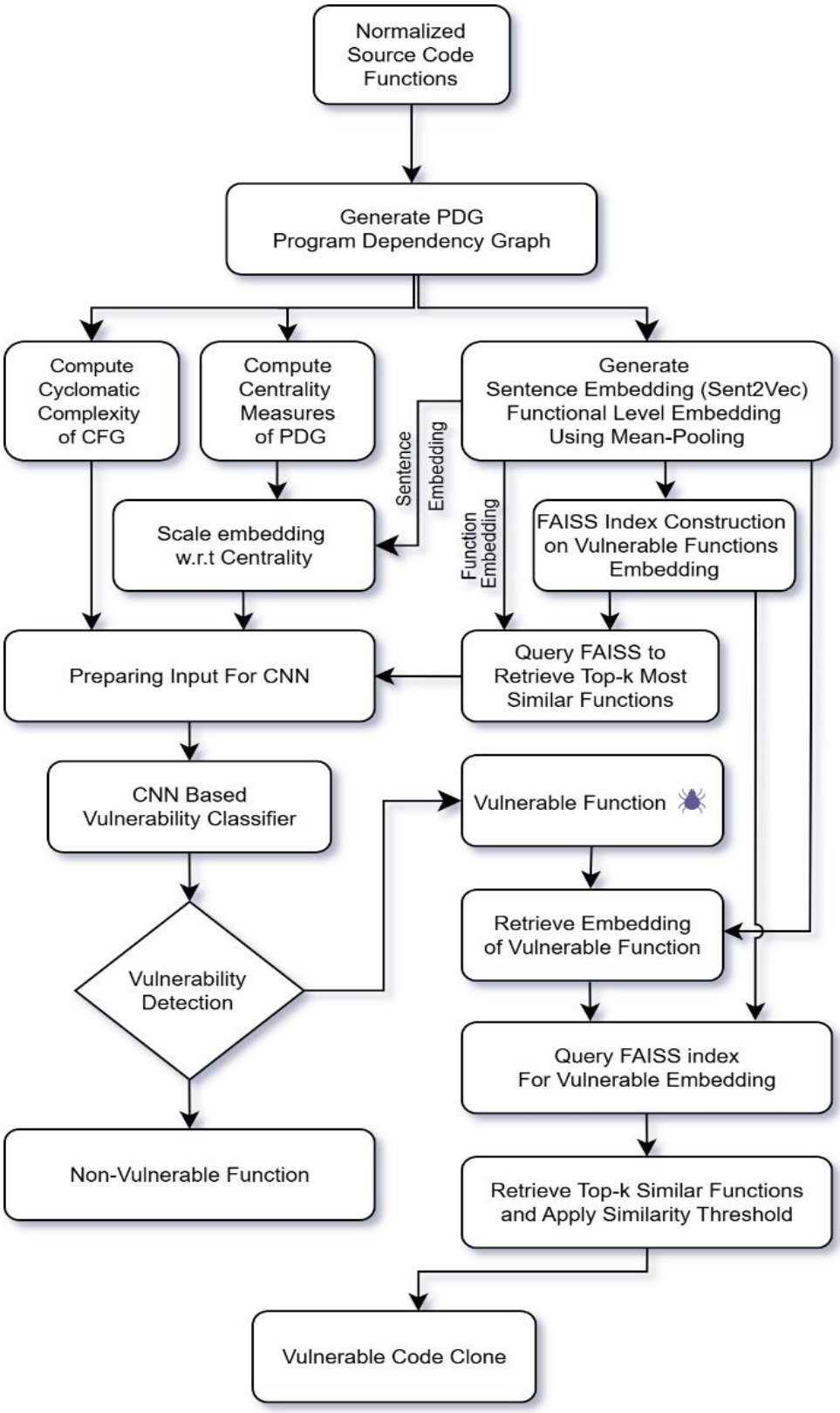


Figure 1: Overview of the AMBER framework and its main processing stages.

Built-in keywords/functions like strcpy directly relate to potential buffer overflow vulnerabilities. For example, CVE-2016-8332 is a heap-based buffer overflow vulnerability due to incorrect use of the strcpy function, allowing an attacker to deploy a denial of service and code execution attack. So, by retaining these built-in keywords, we ensure that potential vulnerability indicators are not lost during the abstraction process. Figure 2 illustrates a code snippet before and after normalisation, a detailed function visualisation at different normalisation steps.

Code Snippet A: Source code of a Function

```
1   // Authenticate user by username/password
2   int auth(char *usr, char *pwd)
3   {
4       char buf[16]; // Temp buffer
5       strcpy(buf, pwd);
6
7       if(strcmp(buf, "Admin@1234")==0)
8       {
9           grantAdm(usr); // Admin rights
10          printf("Hi Admin: %s\n", usr);
11          return 1; // OK
12      }
13      else
14      {
15         printf("Invalid user:%s\n", usr);
16         return 0; // Fail
17      }
18  }
```

Code Snippet B: Removal of Comments, Extra spaces and blank lines

```
1   int auth(char *usr,char *pwd)
2   {
3      char buf[16];
4      strcpy(buf,pwd);
5      if(strcmp(buf,"Admin@1234")==0)
6      {
7         grantAdm(usr);
8         printf("Hi Admin: %s\n",usr);
9         return 1;
10     }
11     else
12     {
13      printf("Invalid creds:%s\n",usr);
14        return 0;
15      }
16  }
```

Code Snippet C: Generalise Function names

```
1   int func1(char *usr,char *pwd)
2   {
3      char buf[16];
4      strcpy(buf,pwd);
5      if(strcmp(buf,"Admin@1234")==0)
6      {
7         func2(usr);
8         printf("Hi Admin: %s\n",usr);
9         return 1;
10     }
11     else
12     {
13        printf("Invalid creds: %s\n",usr);
14        return 0;
15      }
16  }
```

Code Snippet D: Generalise Variable names

```
1   int func1(char *v1,char *v2)
2   {
3      char v3[16];
4      strcpy(v3,v2);
5      if(strcmp(v3,"Admin@1234")==0)
6      {
7         func2(v1);
8         printf("Hi Admin: %s\n",v1);
9         return 1;
10     }
11     else
12     {
13      printf("Invalid creds:%s\n",v1);
14        return 0;
15      }
16  }
```

Figure 2: Source code pre-processing and normalisation

2.2 Graph Construction and Features Extraction

After normalisation, each source code function is transformed into a Program Dependency Graph (PDG) to capture its underlying structural relationships. In this regard, we are using the Joern tool [16] that is a sophisticated code analysis framework extensively used to generate program dependency graphs from normalised program source code. Joern normalises the code and produces the PDG, a representation of the data dependency (dependencies between variables and statements) and control dependency (dependencies from conditional statements like loops and branching) in the code.

We also output quantitative structural features that are indicative of complexity and potentially vulnerability prone patterns after building the PDG using Joern. Specifically, we calculate centrality measures such as degree centrality, closeness centrality, and betweenness centrality to ascertain the prominence and influence of individual nodes in the graph.

• **Degree centrality** $C_D(v)$Measures the importance of a node by counting how many nodes are directly connected to it. A node with higher degree centrality is considered more central and influential. In normalised form, it represents the fraction of all possible nodes that a node is connected to calculated as:

$$C_D(v) = \frac{deg(v)}{n-1} \tag{1}$$

where $\deg(v)$ is the degree of a node v, and n is the total number of nodes in a graph.

• **Closeness centrality** $C_c(v)$ Indicates how fast a node can reach all other nodes in the graph. It is the inverse of the total shortest path distances from the node to every other node. A higher closeness centrality means the node can more easily reach other nodes and is more central.

$$C_c(v) = \frac{n-1}{\sum_{v \neq u} d(v,u)} \tag{2}$$

Where $d(v,u)$ is the shortest distance between nodes, and n is the number of nodes.

• **Katz centrality** $C_K(v)$ is a measure of a node's relative influence in a graph that accounts for all possible connections to it rather than only the shortest paths. The Katz centrality of a node v is defined as:

$$C_K(v) = \alpha \sum_{u \epsilon v} A_{vu} C_{K(u)} + \beta \tag{3}$$

Note that A is the adjacency matrix of graph G with eigenvalues λ. In order for Katz centrality to converge, α must be:

$$0 < \alpha < \frac{1}{\lambda_{max}} \tag{4}$$

λ_{max} is the largest eigenvalue of the adjacency matrix of A.

We will also calculate the cyclomatic complexity of each function, which is the classical metric of control flow complexity. Cyclomatic complexity is calculated with the equation $M = E - N + 2P$, where E is the number of edges, N is the number of nodes in the function's control flow graph, and P is the connected components. Cyclomatic complexity M essentially measures the number of linearly independent paths to traverse through the code. The most complex functions may be more error-prone and contain vulnerabilities such as logic errors or untested corner cases. At the end of Stage 2, we now have each function's PDG, node centrality metrics such as degree, closeness and betweenness, along with cyclomatic complexity. These structural features may suggest vulnerabilities, but further meaningful analysis, including semantic knowledge, will be done in the following stages.

2.3 Semantic Code Embedding and Indexing: In parallel with the structural analysis, Amber captures the semantic features of each function through learned embeddings of the code. We model each function f as an ordered sequence of normalised program statements $\{s_1, s_2, \dots, s_n\}$, where s_i denotes the i^{th} statement. We use a Sent2Vec [17] model trained on the Software Assurance Reference Dataset (SARD), a large corpus of labelled vulnerable and non-vulnerable code examples. Sent2Vec generates fixed-length 128-dimensional vectors for sentences so that semantically similar sentences are close in the vector space. Specifically, we compute:

$$E_i = Sent2Vec(s_i) \tag{5}$$

Where E_i is the 128-dimensional embedding of i^{th}statement. These embeddings provide statement-level semantic representations capturing the intent of individual lines. To create a global representation of each function, we apply mean pooling over the statement embeddings:

$$F = \frac{1}{n}\sum_{i=1}^{n} E_i \tag{6}$$

Where F is the function-level embedding of the current input function.

To incorporate vulnerability context, we construct an index of known vulnerable functions $\{f_v^1, f_v^2, \dots, f_v^n\}$, each associated with function-level embedding F_v^j. For each function F, we compute its Top-k similarity scores.

$$S = \{s_1, s_2, \dots, s_k\}, \quad s_j = sim\left(F, F_v^j\right) \tag{7}$$

Where $sim(.,.)$ denotes the similarity metric computed using FAISS between embedding, FAISS (Facebook AI Similarity Search) is an efficient nearest neighbour search library. These similarity scores populate one of the CNN input channels, providing the model with information on how closely the input function aligns with known vulnerable patterns. The PDG structure is used to support embedding the extraction, making AMBER tightly coupled with both the control flow and execution logic, thus enabling the detection of possible vulnerabilities.

2.4 Feature Fusion and Representation Enhancement: Stages 2 and 3 have provided two parallel sets of features, structural and semantic. In Stage 4, Amber fuses these features to create a single, enhanced function representation suitable as input to a CNN. We design a multi-channel tensor representation that can be interpreted as an image encoding different aspects of the function. First, we prepare the line-level information as a matrix. We take the list of line embeddings E_i from Stage 3, where each E_i is a d-dimensional vector with $d = 128$. If a function has L statements, we can organise the embeddings into an $L \times d$ matrix. For consistency across functions, we set a maximum number of lines to consider $L_{max} = 100$. This value was chosen because most functions in the SARD dataset have fewer than 100 lines. Functions with less than L_{max} are padded with zero vectors to be the length L_{max}, and those functions with more than L_{max} are simply truncated. Hence, we have a 100×128 matrix for each function as a base representation, capturing the semantic content of each line. We then put the structural metrics generated in stage 2 into this matrix. We do that by treating the centrality measures as scaling the line embeddings. That is, we will create a 100×128 matrix for each of the line embedding weighted by a respective centrality measure:

- **Channel 1:** Degree-centrality-weighted embeddings. If $C_D(i)$ is the normalised degree centrality of the i^{th} line, we multiply the embedding E_i by $C_D(i)$thus scaling each component of E_i accordingly. The first channel matrix C_1 has rows defined as
$$C_1[i] = C_D(i).E_i \tag{8}$$
- **Channel 2:** Closeness-centrality-weighted embeddings. Similarly, the second channel is defined as
$$C_2[i] = C_{Cl}(i).E_i\ , \tag{9}$$
Where $C_{Cl}(i)$ is the normalised closeness centrality of node i.
- **Channel 3:** Katz-centrality-weighted embeddings. Let $C_K(i)$ be the Katz centrality of node i, then third channel is defined as
$$C_3[i] = C_K(i).E_i \tag{10}$$
This represents the Katz centrality weighted embeddings. We opted for Katz centrality in place of betweenness in the fused representation to capture global structural influence smoothly.

  These three channels inject structural awareness into the semantic embeddings, if a line is structurally more critical i.e. having higher centrality, its embedding is emphasised. If a line is having low centrality, it is deemphasised. This way, the CNN can potentially focus on important code statements. Now we incorporate the function-level and similarity features:
- **Channel 4:** We will create a 100×128 matrix where we replicate the function-level embedding F, that we computed in stage 3, at all rows. The function-level representation is produced by mean pooling the embeddings of all n statements to produce a general representation of the entire function's semantics. Formally,
$$C_4[i] = F \tag{11}$$
where $C_4[i]$ denotes the i^{th} row and F is the 128-dimensional function-level embedding. This gives CNN a holistic view of the function's semantics at every position. Another way to see this: it allows the CNN's filters to compare each line's embedding from channels 1 to 3 with the overall context of the function, i.e. channel 4 helps to detect if a particular line deviates or if specific patterns are present function-wide.

- **Channel 5:** we will encode the Top-k similarity scores. We will use the first k rows of the channels to encode the Top-k similarity scores. For the i^{th} (with 1-indexing) row of channel 5, we take the scalar value s_i for the similarity score of the i^{th} top match and practice it in all 128 columns. Formally:

$$C_5[i] = \begin{cases} [s_i, s_i, \dots, s_i], i \leq k \\ [0,0, \dots,0], i > k \end{cases} \quad (12)$$

  The remaining rows if $k < 100$, are padded with zeros. This results in channel 5 matrix C_5 of size 100×128 with each of the Top-k rows containing the repeated similarity score s_i, and all subsequent rows are filled with zero for padding. This way, CNN can pick up the magnitude of similarity with known vulnerabilities. It is important to emphasise that these top-k similarity scores are used internally by the CNN as feature signals, leveraging the complete top-k profile embedded in channel 5 to interpret the strength and distribution of external vulnerability similarities to enhance the classification decision. For example, a high value in row 1 of channel 5 indicates that the channel resembles some known vulnerability, which the CNN may determine to be a strong signal for classification. Lower numbered rows are fine details regarding CNN methods; for example, in some methods, the history could be sorted or clustered by type of vulnerability to provide a richer representation.

- **Channel 6:** We dedicate the sixth channel to capturing structural complexity metrics not covered in the earlier channels. One key component is the function's normalised cyclomatic complexity M, which we encode by filling an entire row (for example, the first row) with the complexity value repeated across all columns. this provides a stable baseline demonstrating the global complexity of the function. Because this channel gives the CNN a structural "fingerprint" of the function, for example, a consistent high baseline when the cyclomatic complexity is high, it further enriches the model's ability to capture complexity-related patterns.

Bringing it all together, the function is now represented as a 6-channel image having a size of $6 \times 100 \times 128$. Channels 1 to 3 carry semantic and structural features associated with each function line, channel 4 carries global semantics, channel 5 carries similarity measures, and channel 6 carries structural complexity attributes. There is rich representation because of the different perspectives each channel affords, which is one of Amber's key innovations. Each channel captures some unique aspect of the function and allows the subsequent CNN to discriminate more heuristically than would be possible on either raw code or single-type features. We can formalise the constructions as follows.

$$C_c(100 * 128) = \begin{cases} C_D(i).E_i, & c = 1 \\ C_{cl}(i).E_i, & c = 2 \\ C_K(i).E_i, & c = 3 \\ F, & c = 4 \\ [s_i, s_i, \dots, s_i], & c = 5, i \leq K \\ [M, M, \dots, M], & c = 6 \end{cases} \quad (13)$$

Each C_c is a matrix of shape $100 * 128$ where each $C_c[i]$ is a 128-dimensional vector. All six channel matrices are stacked along the channel axis to form the CNN input vector of shape $6 \times 100 \times 128$.

The motivation behind this multi-channel representation is to enable CNN to learn cross-feature interactions. For example, the CNN could learn a filter that slides across a row (line of code) looking at channels 1-4 together to detect "this line has a high centrality weight in channels1-3 and also matches the global semantic context in channel 4, indicating a critical operation" or a filter might look at a column (embedding dimension) across channels 1-3 to see if certain semantic features consistently appear with structural weights. The channels 5 and 6, mainly being constant or low-frequency spatially, provide background context that the CNN can combine in deeper layers potentially adjusting its confidence if channel 5 shows low similarity to known patterns but channel 6 shows high complexity, suggesting a possible new vulnerability type.

2.5 CNN-Based Vulnerability Detection: After preparing the six-channel tensor, Amber passes this representation to a Convolutional Neural Network (CNN) to classify the function as vulnerable or not vulnerable, and the CNN is the primary learning mechanism of Amber, which learns the patterns of vulnerabilities from the fused representation.

In Amber, we use convolutional filters of size $m \times 128$, inspired by VulCNN such that these convolutional filters span the full width of the embedding. The parameter m determines how many consecutive lines are included in

each filter. In Amber, we use ten filter sizes $m = 1$ to 10 with 32 feature maps per filter size so that the model can extract different patterns in various local and global code regions. The Max pooling layer is applied after the convolutional layer to reduce dimensionality, and the pooled outputs are forwarded to the fully connected layers. We set the fully connected layer length to 320 to match the filter. Amber takes advantage of ReLU (Rectified Linear Unit) for the non-linear activation, and we use cross-entropy loss as the objective function. The Adam optimiser supervises CNN training with a learning rate of 0.001, a batch size of 32, and 100 epochs.

2.6 Vulnerable Code Clone Detection: The last phase of Amber only works on functions already marked as vulnerable by CNN. The important idea is simple: once we detect a vulnerability, we should also check if this same vulnerable code has been copy-pasted elsewhere in the same or related projects. This ensures that developers can fix the detected vulnerability and any hidden cloned vulnerabilities.

To enrich its classification decisions, Amber compares each function embedding F against a pre-built indexed set of known vulnerable function embeddings $\{F_v^1, F_v^2, \dots, F_v^n\}$, which is constructed and indexed during the third step from the Sard and real-world vulnerabilities dataset. This comparison allows Amber to assess whether the analysed function is itself a clone or near-clone of a previously identified vulnerability. If a close match is found, this information is incorporated into the CNN's classification decision and can also be reported to the user. For example, Function f is vulnerable and closely matches CVE-XXXX-XXXX." This step effectively provides external clone detection**.**

For internal clone detection, Amber utilises the function embedding F, which was calculated in Stage 3. Amber also has the embeddings for all other functions in the project $\{F_{g1}, F_{g2}, \dots, F_{gN}\}$, where F_{gi} denotes the embedding for the i^{th} function g_i in the project, and N is the number of functions under consideration. Amber performs a nearest-neighbour search using FAISS in this embedding space, comparing F against all project functions to retrieve the Top-k' most similar functions $\{g_1, g_2, \dots, g'_K\}$, ranked by similarity scores $\{\sigma_1, \sigma_2, \dots, \sigma_K\}$ computed using FAISS.

$$\sigma_i = sim(F, F_{gi}) \tag{14}$$

To decide which candidates, count as vulnerable clones, Amber applies a similarity threshold $\theta = 0.9$ Any function with $\sigma_i > \theta$ is considered a likely vulnerable clone:

If $sim(F, F_{gi}) > \theta$, then g_i is flagged as a clone.

In summary, Amber performs both of these functions:

External clone detection: Checking to see if the function exists in the external index of known vulnerabilities (Stage 3)

Internal clone detection: Finding similar functions internally in itself (Stage 5).

This gives Amber the capability to classify a point of originating vulnerabilities and track its clone origin for cloning vulnerabilities, improving vulnerability coverage overall and assisting its developers in remediating associated issues.

3. Proposed Algorithm

In this section, we present the algorithm used for vulnerable code clone detection. The algorithm takes as input a function-level embedding and structural representations and outputs a classification of vulnerability and clone similarity. It integrates structural centrality, semantic features, and similarity scoring within a CNN-based model. Algorithm 1 and Algorithm 2 together describe how AMBER (Automated Method for Bug and Exploit Recognition), a two-stage framework, works for vulnerability and vulnerable clone detection in source code.

3.1 Vulnerability Detection (Algorithm 1)

The first part of AMBER works on a set of normalized source code functions, denoted by F. For each function f, its Program Dependence Graph (PDG) and Control Flow Graph (CFG) are generated by the algorithm. These graphs are used to compute structural metrics such as Cyclomatic Complexity (M_f), Degree Centrality (C_D), Closeness Centrality (C_C), and Katz Centrality (C_K).

Each statement in the function is then converted into a vector representation using Sent2Vec model. The mean of these embeddings is used to form a function-level representation. This representation is scaled and combined

with replicated centrality metrics, function-level features, and the top-*k* similar vulnerable functions retrieved using FAISS.

Algorithm 1 AMBER: Vulnerability Detection

Require: F: Normalized source code functions
Require: F_{vul_index}: Vulnerable function embeddings database
Require: k: Number of similar functions to retrieve (Top-K)
Require: θ: Similarity threshold for clone detection
Ensure: vul: Set of vulnerable functions
Ensure: v_clones: Set of vulnerable code clone pairs

$vul \leftarrow \emptyset$
$v\_clones \leftarrow \emptyset$
for each function $f \in F$ **do**
    $PDG \leftarrow GeneratePDG(f)$
    $CFG \leftarrow GenerateCFG(f)$
    $M_f \leftarrow CyclomaticComplexity(f)$
    $C_D \leftarrow DegreeCentrality(PDG)$
    $C_C \leftarrow ClosenessCentrality(PDG)$
    $C_K \leftarrow KatzCentrality(PDG)$
    $E_s \leftarrow [\,]$
    for each statement s_i in f **do**
        $E_i \leftarrow \text{Sent2Vec}(s_i)$
        $E_s \leftarrow E_s \cup \{E_i\}$
    end for
    $F_{mp} \leftarrow \text{MeanPooling}(E_s)$
    $X_1 \leftarrow Scale(E_s, C_D)$
    $X_2 \leftarrow Scale(E_s, C_C)$
    $X_3 \leftarrow Scale(E_s, C_K)$
    $X_4 \leftarrow Replicate(F_mp, 100)$
    $S \leftarrow FAISS\_TopK(F_{mp}, F_{vul\_index}, k)$
    $X_5 \leftarrow ReplicateTopK(S)$
    $X_6 \leftarrow Replicate(M_f, 100)$
    $T \leftarrow Stack(X_1, X_2, X_3, X_4, X_5, X_6)$
    $label \leftarrow AMBERCNN(T)$
    if $label == vulnerable$ **then**
        $vul \leftarrow vul \cup \{f\}$
    end if
end for

All features are stacked into a tensor and passed into a Convolutional Neural Network (AMBERCNN), which classifies whether the function is vulnerable. Functions predicted as vulnerable enter a further set for analysis.

3.2 Vulnerable Clone Detection (Algorithm 2)

The second stage of AMBER finds clone pairs among the identified vulnerable functions. For each vulnerable function, a function-level embedding is computed. Thereafter, FAISS is used to retrieve the top-*k* similar functions from the entire function set.

Algorithm 2 AMBER: Vulnerable Clone Detection

Require: vul, F, k, θ
Ensure: v_clones: Set of vulnerable code clone pairs

for each $f \in vul$ **do**
  $F_f \leftarrow FunctionEmbedding(f)$
  $neighbors \leftarrow FAISS\_TopK(F_f, F, k)$
  for each $f' \in neighbors$ **do**
    if $FAISS\_Similarity(F_f, F_{f'}) \geq \theta$ **then**
      $v\_clones \leftarrow v\_clones \cup \{(f, f')\}$
    end if
  end for
end for
return vul, v_clones

Whenever the similarity score between a function and any of its retrieved neighbours exceeds a particular threshold , the corresponding pair is recognized as a vulnerable clone. All such clone pairs are then aggregated into the final output set. θ

For clarity and ease of reference, the notation used in the proposed AMBER framework is presented in following table 1. The listed symbols correspond to graph representations, centrality measures, embedding vectors, feature tensors, and classification variables in vulnerability detection and vulnerable clone detection.

Table 1: Symbols used in the AMBER Algorithm

Symbol	Meaning
F	Set of normalised source code functions
f	A single function from the set F
PDG	Program Dependence Graph of function f
CFG	Control Flow Graph of function f
M_f	Cyclomatic Complexity of function f
C_D	Degree Centrality metric computed from PDG
C_C	Closeness Centrality metric computed from PDG
C_K	Katz Centrality metric computed from PDG
s_i	i^{th} statement in function f
E_i	Sentence embedding of statement s_i using Sent2Vec
E_s	Set of embeddings of all statements in f
MeanPooling(E_s)	Aggregated function-level embedding from E_s
F_{mp}	Mean-pooled function embedding obtained from E_s
X_1, X_2, X_3	Scaled versions of E_s with respect to C_D, C_C, and C_K
X_4	Replicated mean-pooled function embedding F_{mp}
F_{vul_index}	Embedding index of known vulnerable functions used by FAISS
S	Top-k similar vulnerable functions retrieved using FAISS
X_5	Replicated vector of FAISS Top-k results
X_6	Replicated Cyclomatic Complexity vector
T	Final stacked tensor of features passed to the CNN
AMBERCNN(T)	CNN model that classifies vulnerability
vul	Set of functions classified as vulnerable
F_f	Embedding of function f used in Part 2
$neighbors$	Top-k similar functions to F_f retrieved from F
θ	Similarity threshold to determine clone similarity
v_{clones}	Set of identified vulnerable code clone pairs

4. Implementation

We designed Amber using a hybrid of Python-based static analysis tools and deep learning frameworks. The main components and their implementation are as follows:

- **Pre-processing and PDG Extraction:** we used Joern (v1.1.0) to parse C/C++ files and extract PDGs from the C/C++ functions. We created a Python script to process Joern's output, extract the adjacency list from the PDGs, and obtain the cyclomatic complexity. The Centrality measures such as degree, closeness, and katz were Computed using NetworkX and normalised to the [0,1] range.
- **Sent2Vec Embeddings:** We trained a 128-dimensional Sent2Vec model using the SARD dataset. Based on unigrams and bigrams, the model generated sentence embeddings where each normalised code statement with tokens like variable names replaced by generic placeholders was treated as a sentence. This embedding model remained unchanged during Amber's training and was used solely to provide input features to CNN.
- **Known Vulnerability Index:** We populated a FAISS index with embeddings of 12303 known vulnerable functions taken from the SARD dataset. We used cosine similarity, which FAISS handled by L2-normalizing vectors and using inner-product search. We set FAISS to return neighbours for each query. The index enabled sub-millisecond retrieval of nearest neighbours per function embedding on average. K=5
- **CNN Architecture:** We implemented the CNN in PyTorch: the detailed experimental setting and CNN configuration in Table 2 below.

Table 2: Summary of experimental settings and CNN architecture.

Category	Parameters	Setting
Input	Input Vector Size	6×100×128

and architecture	Convolution filter size	$6 \times m \times 128$
	Region size (m)	1 to 10 lines
	Filters per region	32
	Total filters	10 × 32
	Max pooling output	10 × 32
	Fully connected layer size	320
	Activation Function	ReLU
Training configurations	Loss Function	Cross-Entropy loss
	Optimizer	Adam
	Learning rate	0.001
	Batch size	32
	Number of epochs	100
Vulnerable Clone detection	Similarity threshold θ for clone detection	0.95
	Top-$k$neighbours from vulnerability database	5
	Top-k' neighbours from project embedding	10

For a fair comparison, we used the same hypermeters across Amber and VulCNN without performing separate model-specific tuning. This was done to isolate the effect of architectural modification introduced in Amber. Thus, any performance gains reported in Section 5 stem directly from Amber's enriched architecture and multi-channel representation, not from different training setups or tuning advantages. We acknowledge that independent hyperparameter optimisation may further improve the performance for each model, which is left as future work.

- **Training Procedure:** We partitioned the 33,360 functions into training, validation, and test sets using stratified sampling to maintain the same vulnerable-to-non-vulnerable ratio, approximately 37:63 in each subset. In particular, we used 70% of the data for training, 15% for validation, and 15% for testing, i.e. approximately 23,352 functions for training, 5,004 functions for validation, and 5,004 functions for testing. The stratifications ensure that each split replicates the class distributions of the total dataset to prevent it from being too easy or too hard. However, training on raw imbalanced data may induce bias in the model to predict only the majority class, which is non-vulnerable.

To counter this, we employed random under-sampling on the training set. From the 14,800 non-vulnerable training functions, we randomly sampled 12,303 non-vulnerable functions to match the number of vulnerable functions 12,303 in the training set. This yields a balanced training set of 24,606 functions (50% vulnerable). By training on an even class distribution, CNN is encouraged to pay equal attention to vulnerable and non-vulnerable patterns rather than being overwhelmed by the larger class.

- **Threshold for vulnerable clone detection:** We set for high-confidence vulnerable clone marking. This was based on observing that semantic clones, even with Type-3 modifications, often had cosine similarity above 0.97 in our embedding space, whereas unrelated code was usually below 0.85. So, 0.95 was a tight threshold, yielding a few false clone identifications. We chose for internal clone search, meaning for each vulnerable function, we examined up to 10 nearest neighbours for potentially vulnerable clones. $\theta=0.95 K'=10$

5. Experimental Setup

We evaluated Amber on combined datasets: one synthetic and one real-world. The synthetic dataset is generated from SARD, which is a data-set of many small programs containing vulnerabilities. The real-world dataset is composed of functions obtained from open-source projects (GitHub) with identified CVE vulnerabilities and their patched versions, akin to the functions used in previous studies such as Devign[18] and SySeVR [11].

5.1 Dataset Details: A dataset of vulnerable (12,303) and non-vulnerable (21,057) functions was used for analysis, derived from a SARD-based dataset. The dataset covers important vulnerability classes like buffer overflows, null pointer dereferences, use-after-free and command injection. Many examples use the cloned variants so that we can concentrate on vulnerability detection and analysis at the level of functions. We also dug out publicly available

C/C++ code, such as the Apache HTTP Server, OpenSSL, Wireshark, FFmpeg and Linux kernel drivers from 2010–2021 to check for functions related to CVEs.

For fairness the 600 vulnerable functions were matched in size and module to 600 safe functions, either patched versions or functions with no known issues, with CVE or CWE tags. Vulnerabilities and functions are more complex than in SARD [19] for this dataset. It also contains instances of vulnerable clones. For example, a vulnerability found in one project's library function also appears in another project's copy of that library code. We split each dataset into training, validation, and test sets using a 70 percent, 15 percent, and 15 percent division. We ensured that no two functions across the training, validation, or test sets were clones of each other to avoid overly optimistic evaluation. This was especially important for SARD, where many clones exist, so we grouped clones and placed the entire group within a single split.

5.2 Comparison: To evaluate Amber's effectiveness, we compared it against several established baseline techniques using the same experimental settings described earlier, such as class balancing and training hyperparameters. First, we compare with a VulCNN , which is essentially Amber's CNN without the semantic and structural enhancements, relying only on the original three-channel representation of raw code (3×100×128) capturing syntactic and structural features. Next, we included a reproduction of VulDeePecker , a Bi-LSTM model trained on token sequences from code gadgets, and present sequence-based vulnerability detection. We also evaluated a graph neural network model inspired by Devign , applied to our real-world dataset, as Devign was explicitly designed for analysing larger programs using program graphs. Finally, for clone detection evaluation, Amber's post-classification FAISS-based similarity search was compared to NiCad , a widely used clone detection tool. However, unlike Amber, which performs clone detection only on functions classified as vulnerable, NiCad is applied across the full dataset without classification filtering. To enable a fair comparison, we used a manually curated ground truth of known vulnerable clone pairs and applied the same similarity threshold (0.95) to both Amber and NiCad's outputs for evaluating clone detection performance.

5.3 Evaluation Metrics: For vulnerability detection, we report accuracy, precision, recall, and F1 score. We also look at the False Positive Rate (FPR) and False Negative Rate (FNR) since, in security, false negatives are critical to minimise, as missing a vulnerability is generally worse than a false alarm. We plot ROC curves to visualise the trade-off. For vulnerable clone detection, we consider a vulnerable clone pair as true positive if both functions in the pair are indeed vulnerable and correspond to the exact root cause. We then measure the precision and recall of vulnerable clone identification: how many known vulnerable clone pairs were found and how many reported clone pairs were correct. The threshold for classification in Amber was 0.5 by default, but we also analyse performance across thresholds, which is reflected in ROC analysis. The vulnerable clone similarity threshold θ was set to 0.95, as mentioned in the evaluation.

6. Result and Analysis

We present Amber's experimental results on the combined SARD and real-world datasets, comparing its performance to baseline models such as VulCNN, Devign, VulDeePecker, and SySeVR. We present accuracy, precision, recall, F1 score, TPR and TNR metrics. We also assess Amber's a vulnerability clone detection function.

6.1 Vulnerability Detection Performance

Amber was able to achieve a high accuracy of detecting vulnerable functions, exceeding the baseline approaches. The same dataset partitions, input sizes (where applicable) and training parameters used (mentioned in table 2) for training and evaluation of all baseline models, VulCNN, VulDeePecker, and Devign. This made any difference in observed performance due to architectural improvements in Amber alone. The test results on the combined dataset are shown in figure 3 and table 3 as comparison with the baseline VulCNN and other models:

Table 3: Performance Comparison of Vulnerability Detection Models

Technique	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
AMBER	93	90	93	91
VulCNN [21]	88	85	80	82
VulDeePecker [6]	86	83	79	81

Devign [20]	89	87	85	86

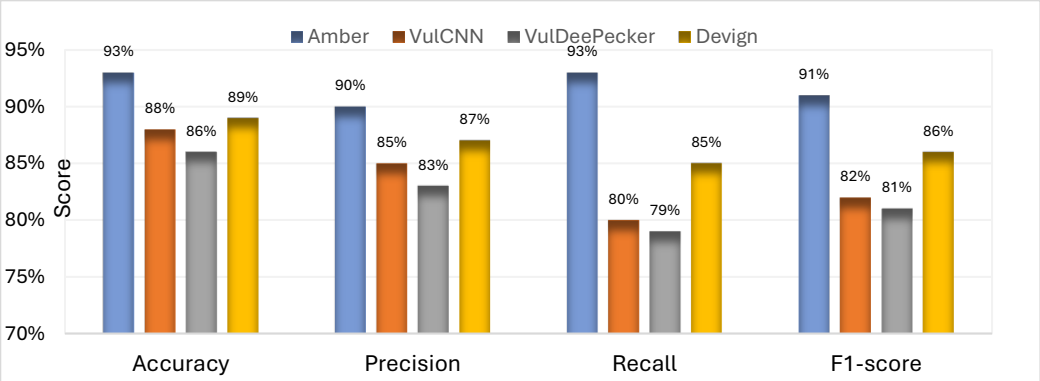


Figure 3: Performance comparison for Amber and baseline models on the combined dataset.

Amber makes some small, but significant, gains on top of the existing baselines. In comparison to the baseline VulCNN model, Amber shows an increase in accuracy of around 5%, recall of around 13% and F1-score of around 9%, indicating the advantage of the extra semantic and structural channels. Amber has a high precision of 90% and a high recall of 93% meaning that she makes few mistakes or false alarms. Because Amber recalled only 7% of vulnerabilities, and VulCNN recalled 20% (recall 80%), this demonstrates that Amber is better at capturing subtle cases, due to its rich set of features. Also, although VulCNN had high precision at low recall, Amber elevated the proportion of recall relative to its precision, resulting in an increase of its F1-score. The improvements over VulDeePecker and Devign are not small, but significantly better, consistently over time, especially when it comes to recall and F1-score. The multi-channel input design of Amber captures a more complete mix of syntactic, semantic and structural features, and these features enhance Amber's performance. Amber's top-K similarity scores, function level embeddings and centrality measures enable it to better identify vulnerable and non-vulnerable patterns than models using one or limited type of features.

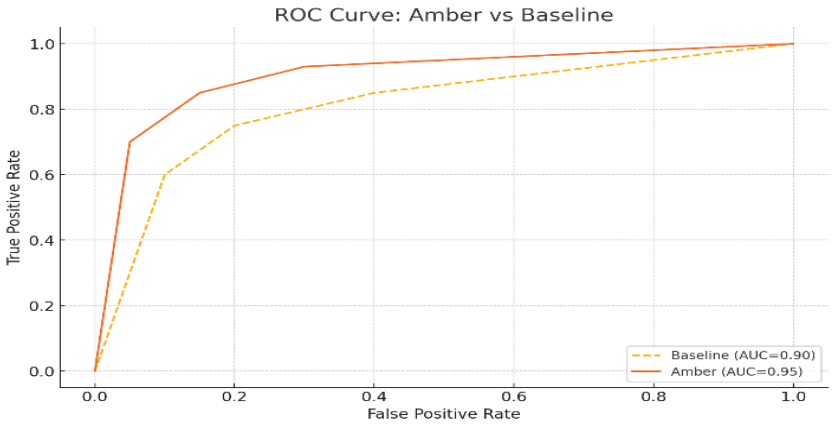


Figure 4: Comparison of Amber with the VulCNN model using ROC on vulnerability detection task

For additional assessment of Amber's performance in different thresholds, the ROC curve is plotted and compared with the VulCNN model on the combined dataset in Figure 4. The solid orange curve is Amber's baseline, and it shows that Amber's true positive rates are more accurate for lower false positive rates. The area under the curve (AUC) for Amber is 95%, and for the baseline is 90%. For example, if the false-positive rate is set at 10% then Amber achieves an 80% true-positive rate, while the baseline is about 70%. This is an example of Amber's increasing vulnerability detection capability at tight operating points, a crucial feature for practical use to minimize false alarms while maintaining high vulnerability detection at tight operating points.

6.2 Vulnerable Clone Detection Performance

To detect vulnerable code clone in the entire codebase, Amber classifies the functions and detects vulnerable functions. This phase was tested using a synthesized set of 600 vulnerable functions from open-source projects for which CVEs have been published, and 12,303 vulnerable functions created with SARD, with the same number of patched and safe functions. Amber searches the entire code base, instead of just vulnerable functions as it filters on function classification, is NiCad. A ground truth set of vulnerable clone pairs was curated for a fair comparison. Filtering and evaluation of the output of Amber's FAISS-based detection and NiCad was performed based on this ground truth with a similarity threshold of 0.95. This integration has practical benefits because in practice Amber does not trigger clone detection for trivial or safe functions, which boosts efficiency and accuracy of clone detection. The criteria for evaluation are as follows.

- **True Positive (TP):** Amber correctly identifies a vulnerable clone pair, a pair of functions marked as clones that are both vulnerable, and share the same root cause flaw. This means that Amber has successfully identified an existing vulnerable code instance that is in another place.
- **False Positive (FP)**: Amber flags a pair of functions as a vulnerable clone when actually it is not. For instance, the code snippets are not connected with one another and could contain no common fault or different faults. In real life, an FP happens when Amber associates a vulnerable function to an unrelated function of a code (coincidentally similar) or associates a vulnerable function to a patched version of the same code.
- False Negative (FN): Amber does not identify an actual vulnerable clone from the dataset. That is: Two functions with the same vulnerability are not recognized. This may be because the similarity of the clones is not sufficient, or because one of the functions was not included in the classification process.

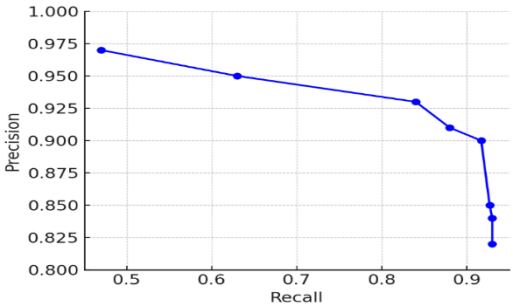
Using these criteria, we evaluated the accuracy, recall, and F1 score of the vulnerable clone detection by Amber. Precision is the ratio of the number of clone pairs reported by Amber that are vulnerable clones; recall is the ratio of the number of vulnerable clone pairs found by Amber to the total number of known vulnerable clone pairs; and F1-score is the harmonic mean of the precision and recall.

To assess the precision, recall and F1-score of vulnerable clone detection we applied these criteria. The results of Amber's clone detection for the combined dataset are summarised in table 4.

Table 4: Amber’s vulnerable clone detection performance on the dataset.

Technique	Precision	Recall	F1-Score
Amber	85	90	87

The clone detection capability of Amber is about 90% meaning that it can detect most of the vulnerable clones that are known. That is, it is about 15% less accurate than roughly 85% and the same code pairs that are flagged are similar but not vulnerable, e.g., safe code or patched code. The score of Amber is 87% (F1) which is a good-mixed coverage and accuracy.



6.3 Challenges in Calculating False Positive and False Negative: with Amber, calculating false negatives and false positives in vulnerable code clone detection has significant difficulties because calculating those requires knowing the ground truth of all true vulnerable clone pairs, which is often not available or incomplete in datasets. To enable the evaluation of false negatives and recall in the vulnerable code clone detection stage, we manually curated a set of verified vulnerable clone pairs within the dataset for the evaluation process. These manually included pairs allowed us to compare Amber’s top-k' clone predictions against known matches, making it possible to measure precision, recall, FP, and FN, even though the original dataset lacked complete Ground truth annotations for vulnerable clones.

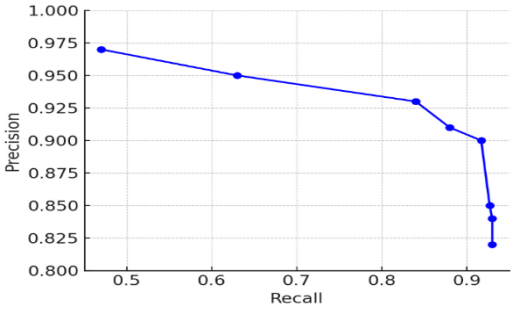


Figure 5: Precision-Recall Curve for Amber's Vulnerable Code Clone Detection

Figure 5 illustrates the trade-off between precision and recall as the similarity threshold varies. At high-recall settings, i.e. 93%, precision declines to the low-80% range, while at high-precision settings 97%, recall drops to approximately 50%. The knee of the curve is around 0.91 recall, and 0.92 precision reflects an optimal operating point balancing detection coverage with minimal false positives. In general, the results indicate that Amber performed well overall on the ability to correctly identify vulnerable code clones.

6.4 Error Analysis: Amber's few false positives frequently resulted from functions performing dangerous operations, but in a safe manner. An example of this is a function that performs numerous pointer arithmetic and buffer operations. Amber marked it as a vulnerable code clone but after further inspection, the function had safe bounds checks. Amber rated it high on the vulnerability probability scale because it appeared to be dangerous. If you can do static analysis checks or post-hoc checks based on a rule that reduces false alarms, those are the things that could help you there; e.g., if you know that an operation involving a buffer write is limited to a known size, you can check it at run-time to exclude a flagged operation.

Some logic errors and very short functions were among the false negative results mentioned. One vulnerability we spotted was a function if (!auth) { privileged_action(); } that would execute privileged_action() if it found that auth was not true. This was a vulnerability for which Amber may not have noticed because it did not appear clearly in the embedding space or training set. Specific vulnerabilities might be those that do not have to do with memory or crashes, but rather logic and policy, such as addition of more features (tracking variables of interest, such as a Boolean auth variable, and more training examples of that pattern). Adding custom analyses such as taint or data flow for authentication flags for such cases could be beneficial.

7. Discussion

Amber shows how just a combination of semantic knowledge, structural analysis and deep learning allows for a potent vulnerability detection system. This section will reflect on what the implications of Amber are, and talk about limitations and future directions.

7.1 Real-World Impact: Amber is meant to serve security auditors and developers working on actual software projects. While noting vulnerabilities and vulnerability clones, Amber eliminates much of the manual work associated with code review and security testing. The two-step output of reporting vulnerabilities and collecting their clones allows for fast mitigation should developers know precisely where to apply a fix after it has been made on the original vulnerability. This is especially valuable in large codebases like operating systems or large frameworks, where a single vulnerable snippet could be copied across dozens of modules, as with the infamous Shellshock vulnerability or various OpenSSL issues copied in IoT projects. Similarly, a tool like Amber could be helpful in continuous integration pipelines to scan for potential vulnerabilities introduced by new commits and notify developers in the pre-release phase. Scanning before release aligns well with a more modern approach to DevSecOps in that problems may be identified sooner.

In addition, Amber facilitates the sharing of knowledge with its embedding index of known vulnerabilities. When new vulnerabilities are discovered and added to the embedding index, Amber can learn from them without re-training the CNN. This process can create a continuous learning cycle within the organisation, such that every incident, or penetration testing discovery, further enhances and refines the organisation's overall detection capability.

A unique benefit is Amber's explainability through clones. The statement "Function X is vulnerable with 95% confidence" can also say "Function X shares characteristics with function Z, and there are several other instances of

similar code in your project" or "it is very similar to function Y, which is known to be vulnerable". This is a good way for developers to understand and to trust the result compared to a black-box model that could be a little more difficult to comprehend. It offers some degree of semantic traceability that is able to track a vulnerability from its first occurrence to all its copies and even trace to known CVE, enabling to understand its impact and choose a remediation approach.

Although Amber has a few strengths there are some things that she has to work around. This will work for the C/C++ language only. Therefore, it cannot be transferred to another ecosystem, such as Java or Python, because these two ecosystems have different vulnerabilities and features, and would need a complete overhaul. While nearly identical to a local structural and semantic context, the PDG and Sent2Vec embeddings do not model inter-procedural dependency and inter-function interactions, and Amber is not able to detect more complex vulnerabilities. Moreover, the precision of the clone detection depends on the similarity threshold which should be set individually on a project depending on the precision/recall criteria.

More than anything else, Amber is designed to complement the current security measures in place, and should be added to a broader security strategy. It can give false-negative and false-positive results during phase. The embedding index needs to be updated and the model re-trained as soon as new patterns of vulnerability emerge if it is to be effective in the long-term.. Finally, the security of the toolchain itself also has to be considered, as embedding vectors could reveal sensitive code under certain circumstances.

7.2 Future Directions: Many promising directions could expand Amber's capabilities. The incorporation of graph neural networks (GNNs) could advance the model by allowing it to use the PDG structure itself together with embeddings. Using contextual embeddings such as CodeBERT or GraphCodeBERT may further enhance the detection of more complex vulnerabilities. Incorporating selective dynamic analysis like concolic testing would allow for verifying flagged issues to remove false-positive issues potentially. A user feedback loop would allow for active learning, allowing Amber to learn project-specific patterns over time. Another promising direction would be expanding the clone detection domain to obtain external codebases, offering promise for discovering cross-project vulnerabilities. Finally, it may also be beneficial to transition from binary vulnerability classification to multiclass classification of vulnerabilities to take advantage of Amber's multi-channelling design to differentiate types of vulnerabilities to provide more defined security observability.

8. Conclusion

Amber is a CNN-based framework for software vulnerability detection, code clone detection and semantic-aware. Amber uses embedding-based clone search to detect vulnerable clones in code bases and leverage a mix of deep learning and program analysis to learn patterns of vulnerabilities well, and to help with more comprehensive remediation. The experimental results indicate that Amber is more accurate and recall robust than baseline methods and demonstrated its capability to find more subtle vulnerabilities as well as previously unlabelled clones. Amber is designed to be generalisable and explainable; useful to vulnerability assessment, secure code review and incident response. Future plans involve expanding Amber to be able to detect inter-procedural vulnerabilities, translate it to other programming languages, incorporate it in developer's workflows, and delve deeper into the possibility of using it for automated code repair. Amber is an important contribution toward intelligent, automation-aided, secure software development.

Declarations

Competing Interests/Conflict of Interest

On behalf of all authors, the corresponding author states that there is no conflict of interest or competing interests.

Funding Information

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Authors' Contributions

[Gurpreet Singh] (0009-0005-1383-8720), [Dhavleesh Rattan] (0000-0002-6295-5078):

These authors contributed equally to this work.

Data Availability Statement

No new dataset was introduced in this study. The experimental evaluation was conducted on the publicly available dataset originally used by VulCNN [20]. The dataset is accessible at https://github.com/CGCL-codes/VulCNN/tree/main/dataset.

Research Involving Human and/or Animals

Not applicable. This study involves only publicly available open-source software repositories and does not involve human subjects or animals.

Informed Consent

Not applicable.

References:

1. "Octoverse 2024: The state of open source | The State of the Octoverse." Accessed: Jun. 03, 2024. [Online]. Available: https://octoverse.github.com/
2. "Heartbleed Bug." Accessed: Jun. 12, 2024. [Online]. Available: https://www.heartbleed.com/
3. 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.
4. N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, New York, NY, USA: ACM, Sep. 2010, pp. 447–456. doi: 10.1145/1858996.1859089.
5. L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CCLearner: A deep learning-based clone detection approach," *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*, pp. 249–260, Nov. 2017, doi: 10.1109/ICSME.2017.46.
6. 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.
7. D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection," *IEEE Trans Dependable Secure Comput*, pp. 1–1, Feb. 2019, doi: 10.1109/TDSC.2019.2942930.
8. Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103184–103197, 2019, doi: 10.1109/ACCESS.2019.2930578.
9. S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep Learning Based Vulnerability Detection: Are We There Yet?," *IEEE Transactions on Software Engineering*, vol. 48, no. 09, pp. 3280–3296, Sep. 2022, doi: 10.1109/TSE.2021.3087402.
10. R. Russell *et al.*, "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, Dec. 2018, pp. 757–762. doi: 10.1109/ICMLA.2018.00120.
11. Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," Jul. 2018, doi: 10.1109/TDSC.2021.3051525.
12. X. Wu, W. Zhao, Z. Tan, X. Zhang, and W. Chen, "Research and Implementation of Code Similarity Detection Technology Based on Deep Learning," in *Proceeding of 2023 9th IEEE International Conference on Cloud Computing and Intelligence Systems, CCIS 2023*, Institute of Electrical and Electronics Engineers Inc., 2023, pp. 235–239. doi: 10.1109/CCIS59572.2023.10262836.
13. 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.
14. C. Zhu, Q. Wang, Y. Tang, and M. Li, "Enhancing code similarity analysis for effective vulnerability detection," in *ACM International Conference Proceeding Series*, Association for Computing Machinery, May 2019, pp. 153–158. doi: 10.1145/3339363.3339383.
15. Y. Zou, B. Ban, Y. Xue, and Y. Xu, "CCGraph: a PDG-based code clone detector with approximate graph matching," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA: ACM, Dec. 2020, pp. 931–942. doi: 10.1145/3324884.3416541.
16. "Joern: Open source robust code analysis platform for C/C++." Accessed: Mar. 25, 2024. [Online]. Available: https://joern.io/
17. M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised Learning of Sentence Embeddings using Compositional n-Gram Features," Mar. 2017, doi: 10.18653/v1/N18-1049.
18. 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
19. "Software Assurance Reference Dataset." Accessed: Apr. 02, 2024. [Online]. Available: https://samate.nist.gov/SARD/test-suites/112

20. Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, “VulCNN: An Image-inspired Scalable Vulnerability Detection System,” *Proceedings - International Conference on Software Engineering*, vol. 2022-May, pp. 2365–2376, 2022, doi: 10.1145/3510003.3510229.
21. 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.