VulChecker: Graph-based Vulnerability Localization in Source Code

Yisroel Mirsky[1], George Macon[2], Michael Brown[3], Carter Yagemann[4]
Matthew Pruett[3], Evan Downing[3], Sukarno Mertoguno[3] and Wenke Lee[3]
[1]*Ben-Gurion University of the Negev*
[2]*Georgia Tech Research Institute*
[3]*Georgia Institute of Technology*
[4]*Ohio State University*

Abstract

In software development, it is critical to detect vulnerabilities in a project as early as possible. Although, deep learning has shown promise in this task, current state-of-the-art methods cannot classify and identify the line on which the vulnerability occurs. Instead, the developer is tasked with searching for an arbitrary bug in an entire function or even larger region of code.

In this paper, we propose VulChecker: a tool that can precisely locate vulnerabilities in source code (down to the exact instruction) as well as classify their type (CWE). To accomplish this, we propose a new program representation, program slicing strategy, and the use of a message-passing graph neural network to utilize all of code's semantics and improve the reach between a vulnerability's root cause and manifestation points.

We also propose a novel data augmentation strategy for cheaply creating strong datasets for vulnerability detection in the wild, using free synthetic samples available online. With this training strategy, VulChecker was able to identify 24 CVEs (10 from 2019 & 2020) in 19 projects taken from the wild, with nearly zero false positives compared to a commercial tool that could only detect 4. VulChecker also discovered an exploitable zero-day vulnerability, which has been reported to developers for responsible disclosure.

1 Introduction

Software vulnerabilities are typically introduced into systems as a consequence of flawed security control designs or developer error during the implementation of the software specification. Such flaws and errors are unavoidable during the design and implementation phases of the software life cycle, particularly for large, complex, and interconnected software systems, such as distributed systems. In 2020, there were 18,352 vulnerabilities publicly reported in the Common Vulnerabilities and Exposures (CVE) database [25], and the total number of new vulnerabilities reported has been increasing every year. In reality, many more vulnerabilities exist and are being silently patched (e.g., [29]), exploited by malicious actors (i.e., zero-days), or simply have yet to be discovered.

Detecting software vulnerabilities in the earliest stages of the software life cycle is critically important, primarily because vulnerabilities in deployed software can be exploited by an attacker to achieve a variety of malicious ends (e.g., data theft, ransom, etc.). Additionally, the cost to patch vulnerabilities at various points in the software life cycle increases dramatically; the cost to patch a vulnerability discovered in deployed software is exponentially higher than the cost to patch the vulnerability during implementation. As a result, detecting vulnerabilities in software source code has been a highly active area of both academic and commercial research and development for decades.

As such, static application security testing (SAST) tools have become pervasive in the software development process to help developers identify bugs and vulnerabilities in their source code. SAST tools typically employ static software analysis techniques (e.g., taint analysis, data-flow analysis) and/or libraries of vulnerability patterns, matching rules, etc., to scan source code, identify potential bugs or vulnerabilities, and report them for manual inspection. SAST reports are highly precise, yielding alerts that identify specific lines of source code as vulnerable to specific types of vulnerabilities, typically categorized as common weakness enumerations (CWEs). Many SAST tools are available to developers, including both open source (e.g., Flawfinder [5], Infer [7], Clang [3]) and commercial offerings (e.g., Fortify [2], Perforce QAC [6], and Coverity [4]). SAST tools are typically employed during code reviews as part of continuous integration and development processes, or launched manually by developers.

Although SAST tools are powerful, they have several disadvantages. First, they often report false positives (i.e., misidentify code as vulnerable). In most cases this is a result of (1) limitations or incomplete rule/pattern matching algorithms, and (2) vulnerabilities that exist in unreachable code. Second, SAST tools struggle to detect complex and contextual vulnerabilities, resulting in high false negative rates (i.e., failing to identify vulnerable code). This is also a limitation of rule/pattern matching algorithms. For example, some program behaviors, such as integer overflows and underflows, are legitimate in some contexts (e.g., cryptography) and vulnerable to exploitation in others (e.g., memory allocation). Similarly, use-after-free vulnerabilities are highly context sensitive, sometimes requiring long sequences of (de)allocations over the program's lifetime to manifest.

1.1 Recent Efforts

To overcome these limitations, researchers have recently turned to deep learning. Specifically, prior work has explored applications of convolutional (CNN) [19, 28, 38], recurrent (RNN) [20–23, 40], and graph neural networks (GNN) [9, 11, 13, 14, 30, 32, 35, 39] to detect vulnerabilities in source code. This is accomplished by representing the software as an annotated graph (G) to encode its data-flow (DFG), control-flow (CFG), and/or abstract syntax (AST).

While prior approaches have demonstrated good performance, software developers are unlikely to adopt or use these models in the same manner they currently use SAST tools. This is because (1) these methods make predictions over entire functions or code regions and/or (2) cannot classify the type of vulnerability found. Using such a tool would require developers to search through hundreds of lines of code for a vulnerability with no further information. This is even more problematic with tools that do not classify the vulnerability [9,11,14,20–22, 30,35,38,39] since the developer has no indication of what they are looking for or what remediation approach is required. This issue is amplified in the presence of false positives, which are common occurrence in existing machine learning techniques.

1.2 Insights

To develop a solution that can be readily used by developers in a comparable manner as existing SAST tools, we first identify the issues with current approaches and their causes:

Broad Program Slicing. To localize a vulnerability, the current approach is to take a subgraph G_i from G as an observation for making a prediction. G_i is either an entire function [9,11,28,30,32,35,39] or a contiguous region extracted from G with a point of interest (PoI) at the center (a.k.a., a program slice) [13, 14, 19, 20, 22, 23, 38, 40]. As a result, a prediction made on G_i can relate to hundreds of lines of code. It also makes it harder for the optimizer since G_i may contain a number of potential root cause and manifestation points.
Insight 1: To increase the precision of a prediction, the observation must relate directly to the location of the vulnerability.

Incomplete Code Representations. To convert G_i into a format that machine learning can understand, symbols derived from the AST and/or functions are either compressed [20–23, 28, 40] or embedded using methods like Word2Vec [9, 11, 19, 30, 32, 35, 39] as a preprocessing step. This method of representing software is compatible with machine learning techniques, but is suboptimal because it prevents the learner from directly reasoning about instruction and dependency semantics. Moreover, information from the AST is either (1) omitted, (2) included in manner that inhibits graph-based learning [9, 11, 30], or (3) is included incorrectly in a manner that harms code semantics and model efficiency [32, 39].
Insight 2: To enable effective graph-based learning in this task, a new code representation that properly incorporates the flow of AST information is required. Furthermore, instruction level semantics should be explicitly provided in G to enable efficient and effective learning.

Manifestation Distance. The manifestation of a vulnerability can occur hundreds of lines after its root cause. Current GNN approaches use models that can only propagate a node's information one or two steps away. As a result, these GNN models cannot infer whether a potential root cause directly influences a potential manifestation point. Instead, these models are limited to identifying local patterns around either point (e.g., a missing guard branch prior to a `memcpy` is enough to raise an alarm).
Insight 3: To increase the model's ability to identify vulnerabilities, the model must be able to identify causality across the entirety of G_i.

The Lack of Labeled Data. Deep learning requires thousands of samples for training. However, since it is hard and expensive to label vulnerabilities in real code at the instruction or even line level, large fine-grained datasets do not exist. Therefore, current work must either: (1) use synthetic datasets (e.g., NIST SARD [26]), which do not generalize to large projects or real world code, or (2) use datasets that only label code regions (program slices) or entire functions.
Insight 4: To enable cost effective high precision detection in source code, there is a need to develop data augmentation techniques to efficiently expand existing datasets.

Level of Program Representation. Source-level programming languages are designed for programmer ease and flexibility, and as such can capture several inter-related and high level operations in a single line of source code. Such rich instruction semantics are in stark contrast to the characteristics of software vulnerabilities, which are typically associated with atomic and machine-level instruction semantics (e.g., out-of-bounds buffer accesses, integer overflows). Most prior work extracts G from the source code without lowering or compiling it. As a result, these models are challenged with identifying low-level vulnerabilities from abstract program representations.
Insight 5: To improve the model's ability to reason about vulnerabilities that have atomic machine-level characteristics, G should capture instruction semantics at lower levels rather than at source code.

1.3 The Proposed Solution

In this paper we propose VulChecker: a deep learning framework for detecting the precise manifestation point of software vulnerabilities in source code. To accomplish this, VulChecker first passes the program's source code through a custom LLVM compiler toolchain to generate a GNN-optimized program representation, which we refer to as an enriched program dependency graph (ePDG).

ePDGs are graph structures in which nodes represent atomic machine-level instructions and edges represent control- and data-flow dependencies between instructions. ePDGs are well-conditioned for graph-based machine learning models since they more accurately capture the flow of the software.

To localize a given vulnerability, VulChecker searches the ePDG for potential vulnerability manifestation points, such as calls to a `free()` function for double free vulnerabilities (i.e., CWE-415). VulChecker then cuts a subgraph G_i *backwards* from that point. By having every subgraph terminate at a potential manifestation point, the model obtains a stable and consistent view. This enables the model to flow information about any potential root causes in G_i down to the singular manifestation point in question. During deployment, when a subgraph is predicted to be vulnerable with respect to a specific CWE, VulChecker alerts the developer to the exact location (line and instruction) of the manifestation point in source code and indicates the CWE type.

In order to give the model the ability to reason about causality, we chose a different GNN than previous approaches. In particular, we develop a model based on a message passing GNN called Structure2Vec (S2V) [15]. With this model, we are able to efficiently pass information over hundreds of hops from one side of G_i to the other. Furthermore, by using S2V we are able to supply edge features (e.g., data type for data-flow dependencies) as well as consider a node's features at every propagation iteration.

Finally, to mitigate the issue of obtaining fine-grained labeled datasets, we propose a data augmentation technique. The approach is to generate positive samples by injecting synthetic traces into projects taken from the wild. The approach is cheap because the synthetic traces are taken from open sourced labeled datasets like the Juliet C/C++ Test Suite [26] and G is manipulated directly to avoid compilation issues. The approach is effective because we inject root causes at random distance away from the manifestation point. This improves generalization by forcing the model to search deeper into G_i over real code.

We evaluated VulChecker on five distinct CWEs: integer overflow (CWE-190), stack overflow (CWE-121), heap overflow (CWE-122), double free (CWE-415), and use-after-free (CWE-416). We chose these CWEs because they are some of the most prevalent and exploitable types of memory corruption plaguing development in popular languages like C/C++. These CWEs cover both spatial and temporal memory safety issues, testing the flexibility of our approach. They are also featured in MITRE's 2021 list of the 25 most dangerous CWEs.[1] When trained on *only* augmented data, VulChecker was able to detect 24 vulnerabilities (reported CVEs) in the 19 C++ projects we evaluated. This was significantly more than the baseline SAST tools, where even the commercial tool only detected 4 of the CVEs. We also had a vulnerability analyst review the top 100 results from a model trained on both augmented and CVE data. The analyst found that VulChecker has hit rate (precision) of 50-80% on the top 50 results. VulChecker was also able to detect a previously unknown vulnerability (zero-day), which has been reported to the developers for responsible disclosure. Overall, we found VulChecker to be a practical tool for developers because: (1) it is able to operate on large projects (such as libgit2 with over 300 files, 110k lines of code, and 18 million instructions), and (2) it has a comparatively low false positive rate.

1.4 Contributions

Overall the main contributions of this work are as follows:

- To the best of our knowledge, VulChecker is the first deep learning framework that can both detect vulnerabilities in source code with instruction and line-level precision and classify the type of CWE. This makes VulChecker the first practical deep learning-based SAST tool for developers, enabling them to quickly identify and remediate vulnerabilities in their code in the same manner as traditional SAST tools. Improving upon prior techniques, developers using VulChecker do not need to search hundreds of lines to find vulnerable code and/or determine what type of vulnerability is present.

- We introduce the use of message passing neural networks for the task of vulnerability detection to learn and identify the relationship (causality) between a vulnerability's root cause and manifestation point in a PDG. Using this model, we are also able to explicitly assign features to edges in our ePDG, such as data type in the DFG component.

- We propose the ePDG, a novel graph-based machine-learning optimized program representation that avoids the inefficiencies and drawbacks of others in prior work.

- We present a novel data augmentation approach that helps mitigate the lack of fine-grained datasets for GNN-based source code vulnerability detection, while still generalizing to real world software projects taken from the wild. The approach is simple and easy to implement, making it a practical way to boost the performance of future models in this research domain.

The source code, models, and datasets presented in this paper are available online.[2] A demo of VulChecker working as a plugin for Visual Studio Code is also available online.[3]

2 Related Works

In this section, we review the last three years of related work on deep learning techniques for detecting vulnerabilities in

[1] `cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html`

[2] `https://github.com/ymirsky/VulChecker`

[3] `https://tinyurl.com/mucpt67x`

Table 1: Summary of related works in comparison with VulChecker

			(1) Code Representation								(2) Sample Selection							(3) Feature Extraction					(4) Model induction				(5) Application				
											Code Slicing																				
			Level		Structure						Plane			PoI		Cut		Node				Edge	Input		Model	Utilizes Edge Type	Detection level				Classifies Vuln. Type?
Year	Cite	Name	Source Code	IR	Linear	CFG	PDG	CPG	ncsCPG	ePDG	Function	Control-flow	Data-flow	Generic	Manifestation	Region	Scoped	One-hot Enc.	Word2Vec	Doc2Vec	Explicit features	Dtype feature	Sequence	Graph			Function	Code Region	Line	Instruction	
2018	[28]	Russle’18	●		●						●					●		●					●		CNN,RF		●				●
2018	[23]	Vuldeepecker	●		●								●	●		●		●					●		BiLSTM			●			●
2019	[40]	μVulDeePecker	●		●								●	●		●		●					●		BiLSTM			●			●
2019	[39]	Devign	●						●		●					●			●					●	GCN,DNN	●	●				
2019	[14]	VGDetector	●			●					●					●				●				●	GCN,DNN			●			
2019	[31]	NW-LCS	●			●					●					●		●						●	LCS Scores		●				●
2020	[19]	Li’20	●		●							●		●		●			●				●		CNN			●			●
2020	[38]	Zagane’20	●		●							●		●		●		○					●		DNN			●			
2020	[32]	Funded	●						●		●					●			●					●	GNN,GRU	●	●				●
2020	[30]	AI4VA	●					●			●			●		●			●					●	GNN,GRU	●	●				
2021	[22]	SySeVR	●		●							●		●		●		●					●		BiRNN			●			
2021	[20]	Li’21		●	●							●		●		●		●					●		CNN+RNN,DNN			●			
2021	[21]	Vuldeelocator		●	●							●			●	●		●					●		BiRNN				●		
2021	[13]	DeepWukong	●				●					●		●		●				●				●	GCN,DNN			●			●
2021	[35]	Wu’21	●				●				●					●			●					●	GNN,DNN	●	●				
2021	[9]	BGNN4VD	●					●			●			●		●			●					●	GNN,GRU	●	●				
2021	[11]	Reveal	●					●			●			●		●			●					●	GCN,DNN	●	●				
		VulChecker		●						●		●			●		●	○			●	●		●	GN (S2V)	●			●	●	●

source code and contrast them to VulChecker’s model, program representation, and application (summarized in Table 1).

2.1 Model

Early work in this area focused on CNNs and RNNs on linear (sequential) representations of the software’s data-flow graphs [19, 23, 28, 38, 40]. However, a linear representation omits software structure, which prevents the model from learning and utilizing various contexts and semantics in its pattern recognition. These linear reductions also make it hard for the model to perform well on code from the wild where a vulnerability can manifest after hundreds of lines of noisy and complex code [11].

To overcome these issues, later work [14, 39] [9, 11, 13, 30, 32, 35] used graph-based neural networks to consider the code’s structural semantics. They utilize a graph convolutional neural network (GCN) that propagates information to neighboring nodes to learn embeddings for each node, which are then averaged or summed prior to use in classification. However, GCNs do not recall (i.e., propagate) a node’s original feature vector at each iteration and struggle to learn long-distance relationships across the input structure. To remedy this, subsequent work [9, 30, 32] used gated graph recurrent neural networks (GRNN) that leveraged a recurrent layer to recall information passed to neighboring nodes at previous iterations. However, in these networks the number of layers dictated the number of propagation iterations, which was only one or two [13]. VulChecker’s model is based on Structure2Vec (S2V) [15], a message-passing neural network that can perform hundreds of iterations without needing additional layers. Consequently, VulChecker is better suited for vulnerability detection because it can identify connections between distant root cause and manifestation points.

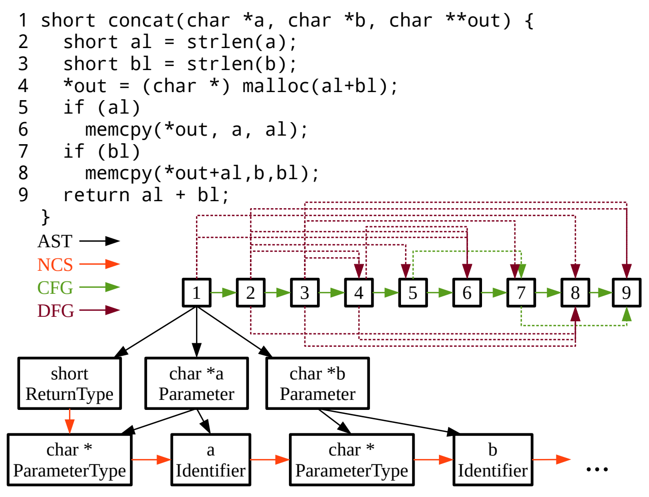
2.2 Program Representation

Prior work used a variety of graph-based program representations (depicted in Figure 1), the simplest of which are control-flow graphs (CFG), data-flow graphs (DFG), and combined CFG/DFGs called program dependency graphs (PDG). These representations can be readily generated from source code (e.g., with tools like Joern [37]), but do not capture instruction semantics explicitly.

More advanced representations include code property graphs (CPG) [37] and natural code sequence CPGs (ncsCPG) such as in [39]. CPGs merge the source code’s PDG and AST by attaching each source line’s AST subtree under the corresponding node in the PDG. This indirectly captures instruction semantics along with control- and data-flow dependencies in a single structure. However, such representations are not well structured for GNN learning models since information from the AST cannot flow across the graph. Finally, an ncsCPG is a CPG in which the leaf nodes in each AST subtree are superficially linked to leaf nodes in the preceding or successive trees, as visualized with orange edges in Figure 1 [32, 39]. This enables information to flow from the AST during GNN learning, however it is suboptimal because semantically unrelated source code lines become linked, even if they share no data or control dependencies.

Finally, systems that use sequential models (i.e., CNNs and RNNs) represent code as a sequence extracted or flattened from one of the graphical forms above [19–23, 28, 38, 40]. While appropriate for the model type being used, these representations fail to leverage the inherently graph-like structure and operation of software.

Regardless of the program representation used, prior work relies on feature extraction to compress and express the contents of each token or node in G. Examples include one hot encodings and pre-processed embeddings (e.g., Word2Vec [24]) to capture the meaning of different symbols and calls (e.g., `int`, `=`, `for`, `free()`, etc.) In some cases entire portions of code are summarized using Doc2Vec [18]. The issue with these representations are that (1) nodes in G_i would likely capture multiple



```
1 short concat(char *a, char *b, char **out) {
2   short al = strlen(a);
3   short bl = strlen(b);
4   *out = (char *) malloc(al+bl);
5   if (al)
6     memcpy(*out, a, al);
7   if (bl)
8     memcpy(*out+al,b,bl);
9   return al + bl;
  }
```

Figure 1: A comparison of different code representations. Most prior systems rely on CFG (green) and DFG (purple) edges alone, while more recent work includes ASTs (black) to capture the code better. However these trees are not conducive to graph-based learning due to their terminal leaf nodes. This is mitigated by connecting leaves with natural code sequence (NCS) edges (orange) [32, 39], but this results in arbitrary paths w.r.t. the data and control dependency edges.

operations in a single line of source code resulting in a loss in semantic precision and (2) the use of pre-processed embeddings prevents the model from learning the best representation to optimize the learning objective (i.e., classifying vulnerabilities). In contrast, our proposed ePDG explicitly defines node and edge features that encode relevant information (e.g., operations) while avoiding superficial features (e.g., variable names).

2.3 Application

Some recent works have proposed detecting vulnerabilities at the binary level. For example, in [36] the authors propose Gemini, a deep learning model which uses Structure2Vec and a Siamese network to measure the similarity of a given function's CFG to that of a vulnerable one at the binary level. Other approaches such as DeepBinDiff [17], ASM2Vec [16], and Bin2Vec [8] all follow similar approaches for vulnerability detection where a repository of compiled vulnerable code snippets are checked against the binary in question. However, these approaches are limited since (1) the models search for instances of vulnerabilities (e.g., heartbleed) and not the general pattern (CWE), (2) they require binary disassembly, a process that is undecidable in the general case, and as such may miss bugs in code the disassembler fails to recover, and (3) the vulnerabilities identified using these methods indicate code regions and not specific lines.

Regarding vulnerability detection in source code, the related works listed in Table 1 cannot directly identify the line of a vulnerability because their representations of G_i do not anchor a specific code line. Instead, they cut graphs to include an entire function [9, 11, 28, 30–32, 35, 39] or the code surrounding a potential root cause point for any vulnerability [13, 14, 19, 20, 22, 23, 38, 40], which leads to very broad predictions over entire functions or larger code regions.

One exception is VulDeeLocator [21], a work developed in parallel to VulChecker. In this work the authors first find PoIs in the source code through the program's AST and mark all library/API function calls, array definitions, pointer definitions and arithmetic expressions. Each marked PoI is then traced down to a lower level code representation (LLVM IR).[4] Then, a forward and backward program slice is taken around the PoI and the slice is flattened into a sequence of 900 tokens (e.g., `call`, `void`, `@`, `FUN1`, '(', ...). Next, each token in the sequence is embedded into a vector using Word2Vec and the sequence of vectors is passed to a bidirectional recurrent neural network (biRNN) which predicts which line is vulnerable.

However, similar to prior work [9,11,14,20,22,30,35,38,39], VulDeeLocator cannot indicate the vulnerability being detected. This leaves the developer to guess the vulnerability out of hundreds of CWEs. Moreover, selecting PoIs from ASTs is not suitable for spatial and temporal memory safety violations. In contrast our ePDG representation based on IR is closer to machine code and resolves ambiguities regarding data types, temporary variables, and storage locations. Furthermore, like other works, VulDeeLocator remove loops and flatten code into a sequence into a finite number of tokens which reducing the code's semantics and patterns. Finally, many source code based systems like VulDeeLocator tailored to specific languages. However, VulChecker performs analysis at the IR level, making it somewhat language agnostic. Although we only evaluate on C and C++ projects in this paper, VulChecker has the potential to work on other languages such as Objective-C, Fortran, and Rust since LLVM can lower them as well. Further research is need to verify compatibility.

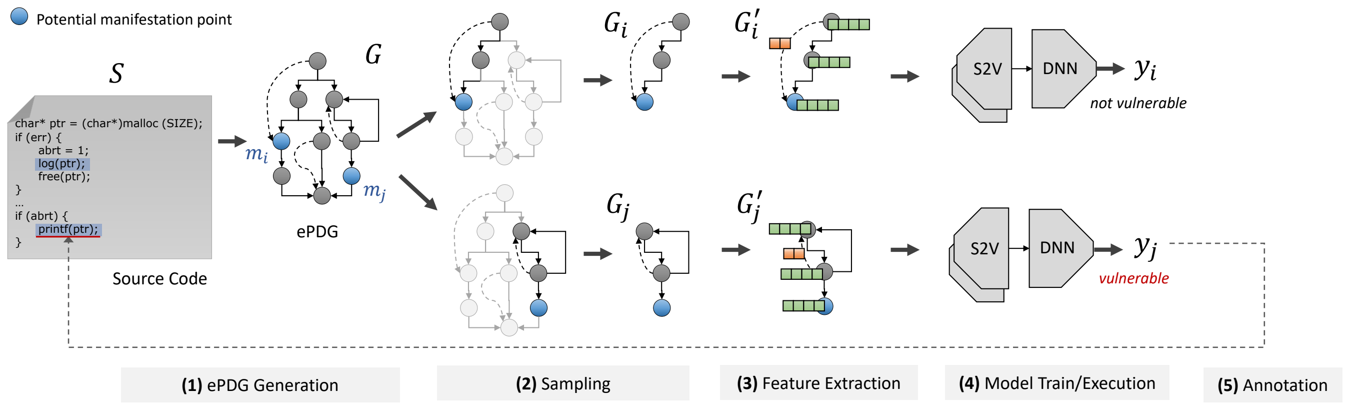
In summary, in contrast to the current state-of-the-art, VulChecker (1) can locate vulnerabilities at both line and instruction level, (2) can classify the type of vulnerability, (3) can better associate root cause and manifestation points by reaching deeper into program slices using a message passing GNN, and (4) may be generalized to a wider array of programming languages.

3 VulChecker

This section presents details on how the VulChecker framework functions. We start with an overview of the pipeline phases and important notation and then elaborate on each step.

Overview. For each CWE (i.e., class of vulnerability), VulChecker trains a separate model and uses a different sampling strategy to extract observations (i.e., potential manifestation points). Each of VulChecker's CWE pipelines follow the same four steps: (1) ePDG generation, (2) sampling, (3) feature extraction, and (4) model training or execution.

[4] `https://llvm.org/`



```
char* ptr = (char*)malloc (SIZE);
if (err) {
    abrt = 1;
    log(ptr);
    free(ptr);
}
...
if (abrt) {
    printf(ptr);
}
```

Figure 2: A diagram showing the steps of VulChecker's pipeline for one CWE. Note that the real graphs are significantly larger than what is visualized (e.g., projects like `libgit2-v0.26.1` have over 18 million nodes in G). Solid edges represent control-flow and dashed edges are data dependencies.

Figure 2 provides a visual summary of these steps.

1. **ePDG Generation:** Given a target program's source code (denoted by S), we first compile it to LLVM IR (intermediate representation) and apply several optimizations. Next, we use a custom LLVM plugin specific to the CWE type of interest to analyze the IR, tag potential root cause and manifestation points, and produce an ePDG of S denoted by G. This process is detailed in Section 3.1.
2. **Sampling:** Next, we scan G to locate potential manifestation points for the given CWE (Section 3.2) that were tagged during ePDG generation. A manifestation point is any node m (instruction) in G that is known to manifest that CWE (e.g., stack memory writes for stack overflow). For the i-th manifestation point in G, we cut a subgraph G_i backwards from that point up to a given depth. G_i represents an observation upon which VulChecker predicts whether or not a vulnerability exists.
3. **Feature Extraction:** Using the same structure of G_i, we create an annotated graph G_i' where each node contains a feature vector that explicitly captures the respective instruction in G_i. Similarly, we add explicit features to the edges of G_i'. Consequently, G_i' captures the code leading up to the i-th manifestation point in a manner that a graph-based machine learning model can understand (Section 3.3).
4. **Model Training/Execution:** VulChecker predicts whether the i-th manifestation point is vulnerable or not by passing G_i' through a S2V model M (Section 3.4). The S2V model is a binary classifier trained on a specific CWE. To train M, we use many negative samples of S, where each sample has been augmented with many positive synthetic examples (Section 4).
5. **Annotation** If M predicts G_i' as positive, then the debug symbols passed down from S to the i-th manifestation point in the ePDG are used to report line number and instruction to the developer.

3.1 ePDG Generation

The creation of an ePDG consists of two steps: (1) lowering the source code S to LLVM IR and (2) extracting G based on the structure and flows it contains.

3.1.1 Lowering Code to LLVM IR

The first step in ePDG generation is compiling the source code S to LLVM IR, which provides a machine-level representation of the target program. This process greatly simplifies the program representation with respect to control-flow (e.g., complicated branching constructs in source code are reduced to conditional jumps that test a single condition), data-flow (e.g., definition-use chains are shorter and less complex as they are based on virtual register values rather than source code variables), and program semantics (IR instructions are atomic and directly translatable to instruction set architecture opcodes). To perform the initial lowering of source code, VulChecker uses LLVM's C/C++ frontend, Clang (v11.0.0). During lowering, VulChecker instructs Clang to embed debug information in the IR, which enables traceability of IR instructions back to source code instructions. If S contains labels, then they are propagated to the LLVM IR using LLVM-provided debug information. These labels are later passed down to ePDG nodes at a later stage.

In addition to simplifying the program representation, VulChecker also uses semantic-preserving compiler optimizations provided by LLVM to simplify and better express the code in G. Specifically, it applies: (1) function inlining to replace function call sites in the IR with a concrete copy of the called function body, (2) indirect branch expansion to eliminate indirect branching constructs, and (3) dead code elimination to reduce the size of the output graph. All together, these optimizations ensure that VulChecker can efficiently analyze our ePDGs interprocedurally in the domain of the original program, as the ePDGs contain fully-precise control- and data-flow information in a minimally sized, connected graph.

3.1.2 Generating the ePDG

With the simplified version of S in LLVM IR form, the next objective is to generate an ePDG, G, that captures the target's control-flow, data-flow, and instruction semantics with minimal loss. Concretely, G is a multigraph defined as:

$$G := (\mathcal{V}, \mathcal{E}, q, r) \tag{1}$$

where $\mathcal{V}$ is a set of nodes (instructions), $\mathcal{E}$ is a set of edges (flows), and both q and r are mappings of nodes and edges to classes and attributes. More formally, let q be a mapping of nodes in $\mathcal{V}$ to a class of instruction, defined as

$$q : \mathcal{V} \to \{\{c, a\} : c \in C, a \in A_c\} \tag{2}$$

where C is the set of all types of instructions in the LLVM instruction API (e.g., `return`, `add`, `allocate`, etc.) and A_c is the set of all possible attributes for instruction $v \in \mathcal{V}$ of type c. Examples include static values in arithmetic operations, function names for `call` instructions, and root cause and manifestation tags. Let r be a mapping of edges in $\mathcal{E}$ to a pair of nodes defined as:

$$r : \mathcal{E} \to \{\{(x, y), d, b\} : x, y \in \mathcal{V}, d \in D, b \in A_d\} \tag{3}$$

where D is the set of edge types (i.e., control-flow or data-flow) and A_d is the set of flow attributes for a flow type d (e.g., the data type of the data dependency).

To generate G according to this definition, VulChecker uses a custom plugin for LLVM's middle-end optimizer, Opt (v11.0.0). This plugin first invokes LLVM's built-in control- and data-flow analyses, and then performs an instruction-by-instruction scan of the target code. For each instruction I_j, VulChecker creates a corresponding node $v_j \in \mathcal{V}$ and mapping $q_j \in q : v_j$. Next, VulChecker uses LLVM's API to extract semantic information about I_j to populate the entry $\{c_j, a_j\}$ of q_j (e.g., operation, if the instruction is a conditional branch, etc.). In addition to semantic information obtained directly from LLVM's API, these attributes also include debug information (e.g., source file and line number), tags indicating potential root cause and manifestation points, and labels indicating actual root cause and manifestation points for model training.

Next, VulChecker performs a second instruction-by-instruction pass to generate control- and data-flow edges in G. This is determined using LLVM's API for identifying an instruction's predecessors/successors and values defined/used. For each of a given instruction's predecessors and successors, a corresponding edge $e_{j,k} \in \mathcal{E}$ is generated with the appropriate type and attributes. Control-flow edges are assigned a `void` data type, and data-flow edges are assigned a data type corresponding to the value definition in the origin node (I_j). After both passes over the IR instructions derived from S finish, the ePDG generation is complete. VulChecker outputs G in JSON format for the next step: sampling.

3.2 Sampling

Now that S is lowered into G, VulChecker can extract observations from G for the machine learning process. First, we identify the points of interest (PoI) in G, which are all potential manifestation points for a given CWE (denoted $m_i \in \mathcal{E}$). VulChecker then cuts the subgraph G_i from G such that m_i is anchored in G_i as the termination node.

3.2.1 PoI Criteria

To extract samples from G, we first identify the nodes in $\mathcal{E}$ where the given CWE can manifest itself. We identify a PoI, m_i, using a lookup list of potential manifestation IR instructions. This list is curated based on our domain expertise and related work. Specifically, for integer overflow (CWE-190), a PoI is any call to a function that passes integer arguments.[5] While we acknowledge that this is a heuristic, it is well validated in prior work [33] to be an accurate criteria for distinguishing intended and unintended overflows. This is the only heuristic we use in our system. For stack and heap overflow (CWE-121,122), the PoIs are any store instructions to local or dynamically allocated memory, respectively. For use-after-free (CWE-416), PoIs are any memory accesses to dynamically allocated memory and for double free (CWE-415), any calls to the memory manager's free function. Aside from CWE-190, these criteria are conservative, ensuring that all true positive manifestation points will have a corresponding PoI (m_i) in G.

For double free, notice how our design assumes that we already know which function in the program performs freeing. In practice, this information can be provided by a developer; our system's intended end-user.

3.2.2 Program Slicing

For each potential manifestation point m_i identified in $\mathcal{E}$, VulChecker crawls G backwards from m_i using breadth first search (BFS), up to a predefined depth n_{depth}, where n_{depth} is a user defined parameter of VulChecker.

The resulting subgraph G_i efficiently and effectively captures any guarding branches and root causes that may result in a positive or negative manifestation of m_i. This is because the BFS is performed over *both* control-flow and data-flow edges, indiscriminately. Consequently, instructions that may be far apart in terms of control-flow (i.e., intermediate basic blocks) can be closely linked by data-flow, which is useful for bug classes like use-after-free.

Finally, since m_i is the termination node of G_i, the manifestation point in question is anchored to a static location, benefiting effective message passing and localization of the prediction—by obtaining the node's metadata $q(m_i)$.

3.2.3 Labeling

For each G_i extracted from G, we associate a label $y = \{\text{negative, positive}\}$ based on the ground-truth for the terminating manifestation point m_i in G_i. Positive labels (vulnerable) are assigned to lines in the source code (for convenience) and then mapped down to IR using debug symbols. If a line of source code contains more than one potential manifestation point, we apply the label to the last

[5]Example: `int x = y * 5; return foo(x);`

relevant IR instruction in the statement.[6] Conversely, any m_i not labeled as vulnerable receives a negative label in G.

When labeling code projects taken from the wild, it is not possible to have the complete ground truth for all the vulnerabilities (exploitable m_i) contained in the code because there might be zero-day bugs. In practice, this means that our training labels may contain some false negatives. However, we observe good performance from our system in practice, which leads us to believe false negatives are rare and of minor consequence. We also note that correctly labeled negative samples will significantly outweigh[7] mislabeled zero-day bugs in the training set, and deep neural networks are robust to noisy labels [27].

Finally, we clarify that VulChecker obtains its positive samples from synthetic datasets. Therefore, labeling of vulnerabilities in projects from the wild is only necessary to evaluate VulChecker (good performance can be obtained from augmented datasets described in section 4).

3.3 Feature Extraction

For each G_i extracted from G, we make a machine learning observation $G_i^{'}$, which has the same graph structure as G_i. The nodes and edges of $G_i^{'}$ are assigned feature vectors that express the respective instructions and flows. Table 2 summarizes the feature vectors, where 'count' is the number of features. There are a total of 1352 node features and 8 edge features. A node's feature vector captures the respective instruction using operational, structural, and semantic features. Edge features capture information regarding connectivity (control and data flows).

Operational Node Features. To capture the operation performed at a node (instruction), we extract features such as the static value, the operation type, the basic function, and whether or not the instruction is part of an `if` clause. The static values are important for recognizing guard checks implemented by the developer to gracefully prevent our target vulnerability classes. Knowing whether an instruction is part of an `if` clause is also a helpful context to identify bug-preventing checks or rare cases where a bug could manifest. The operation and function features are one-hot encodings that represent the action of a node (instruction). The categories for these features were collected by scanning several large software projects from the wild (over 40 million nodes). Some of the operations include: `add`, `load`, `store`, `allocate`, `unsigned_divide`, `logical_shift_right`, `get_element_pointer`, and so on. Some of the functions include: `malloc`, `free`, `fmemopen`, `printf`, and `lseek`, among others. We also reserve a category called 'other' for operations which did not appear in the code collection, but may appear in training and testing. We chose to include an 'other' category so that the network can apply a weight to these occurrences. To encourage the model to learn the patterns and avoid bias (cheating) we set the function feature of m_i to zero. We note that since VulChecker inlines all user functions during the ePDG lowering process and use an IR representation, there is no need to tokenize function names like in prior work. Therefore, the features of $G_i^{'}$ explicitly represent the information that best benefits the learning process.

Structural Node Features. Since $G_i^{'}$ is a directed graph, we use graph features to help the model understand the influence of each node as it propagates messages though the structure. For example, knowing the distance from the nearest potential root cause point r to the anchored potential manifestation point m_i gives the machine learning model implicit information on where relevant nodes are in the graph. We identify a potential root cause point r_j in G_i using domain expertise. Namely, integer arithmetic operations are potential root causes for integer overflow, stack and heap writes for stack and heap overflow, respectively, and calls to free memory for use-after-free and double free. We also use a feature called the betweeness centrality measure (BEC) [34]; a score that measures how critical a node is for efficiently drawing paths between nodes in $G_i^{'}$. This is a common feature used in graph-based machine learning, and graph-based anomaly detection [12]. We use this feature because it provides the S2V model with implicit information on a node's relative location in $G_i^{'}$. Later in section 3.4 we will explain how S2V uses these features to make decisions on how to pass messages across the for graph classification.

Semantic Node Features. In addition to marking the distance to the nearest root cause point, we also indicate if a node itself is a potential root cause or manifestation point. This helps the model locate potential vulnerability sources to propagate to m_i and other manifestation points that may be absorbing the signal prior to m_i. We also note the output type of the node's operation (e.g., `int`).

Edge Features. For edge feature vectors, we indicate the type of edge (control-flow or data-dependency) and capture the data type of the data-dependencies so that the model can capture what kind of data goes where. By knowing the flow of static values, external inputs (from certain functions), and their data types, the model has enough information to foresee (simulate) the impact of data on a program.

We represent the attributed graph $G_i^{'}$ as the tuple (X_v, X_e, A, C): a matrix of node features, X_v, a matrix of edge features, X_e, its adjacency matrix, A, and its incidence matrix, C.

3.4 Model Training & Execution

VulChecker's machine learning model consists of two components that are trained end-to-end: a graph embedding network M_G and a deep neural network (DNN) classifier M_C. Our graph embedding network M_G is a adaptation of the Structure2Vec (S2V) model from [15]. The model uses a neural network to generate node embeddings by passing messages across the graph's structure. The parameters of M_G are fitted to M_C's classification learning objective such that $M_C\left(M_G(G_i^{'})\right) = y$ where y is the probability that m_i is a vulnerability.

The execution of $M_G(G_i^{'})$ consists of a number of iterations of message passing from each node $v_i \in \mathcal{V}$ to its neighbors $v_j \in \Gamma(v_i)$, where a message from node v_i is in the form of the

[6]Example (integer overflow): `int x = (y * 5) + z;`

[7]A project from GitHub can have over 100k negative samples.

Table 2: Summary of Features used in $G_i^{'}$

	Name	Type: Bool	Type: Num.	Type: Categ.	Count
Vertex	Has static value?	•			1
	Static value		•		1
	Operation {+, *, %, ...}			•	54
	Basic function {malloc, read, ...}			•	1228
	Part of IF clause	•			1
	Number of data dependents		•		1
	Number of control dependents		•		1
	Betweeness centrality measure		•		1
	Distance to m_i		•		1
	Distance to nearest r		•		1
	Operation of nearest r			•	54
	Output dtype {int, float, ...}			•	6
	Node tag {r, m, none}	•			2
				Total	1352
Edge	Output dtype {float, pointer ...}				6
	Edge type {CFG, DFG}				2
				Total	8

vector (embedding) e_i. At the start of each iteration, a neural network is used to predict what the next broadcast message of the i-th node should be, based on (1) its neighbors' last messages and the feature vector stored in that node ($x_{vi} \in X_v$) and incident edges ($x_{eij} \in X_e$). This is formulated as

$$e_i = \text{ReLU}\left(W_v x_{vi} + \sum_{j \in \Gamma(i)} W_e x_{eij} + \sigma\left(\sum_{j \in \Gamma(i)} e_j\right)\right) \quad (4)$$

where W_v and W_e are matrices whose parameters are learned during training, and σ is a deep neural network. A single iteration can be computed in matrix form as $E_t = \text{ReLU}(W_v X_v + C W_e X_e + \sigma(A E_{t-1}))$, where E is a matrix containing the current node embeddings. Following the suggestions of [36], we use the ReLU activations in σ to help model complex relationships in the graph.

After n_{iter} iterations (a user parameter), the node embeddings in E are averaged together to form a single embedding vector that is passed through a batch normalization layer before being passed to M_C.

To train the model parameters of M_G and M_C, we optimize the following learning objective function:

$$\min_{W_v, W_e, \sigma_{M_G}, \sigma_{M_C}} \sum_i \mathcal{L}_{CE}\left(M_C(M_G(G_i^{'})), y\right) \quad (5)$$

where L_{CE} is the standard cross-entropy loss function.

Because of the different feature sets, we train separate model for each CWE. We argue that having multiple models (one for each CWE) is reasonable since (1) it only takes 1-2ms to execute a model (2) other SAST tools (like Perforce QAC and Checkmarx) also require the user to select which CWEs to scan for since not all CWEs are important to the developer and each pattern adds complexity to the search, and (3) the models are very small (250KB each) making them very easy to store as well as execute well on a non-GPU system.

3.5 Hyperparameters

Aside from the network parameters (e.g., depth and width of M_C and M_G), VulChecker has two main hyperparameters: n_{depth}, n_{iter}. Parameter n_{depth} cuts the subgraph backwards from a potential manifestation point (m_i), in hopes to include a potential root cause. Parameter n_{iter} controls how far information can be shared across the subgraph, in hopes to correlate the direct influence of potential root causes on m_i. Regarding time performance, increasing n_{depth} has a polynomial time complexity depending on the branch factor of G, whereas increasing n_{iter} has a linear impact. Regarding task performance, setting n_{depth} too large harms performance since more irrelevant information is included in G_i. On control-flow edges, it is possible that the root cause for a positive m_i will be further than n_{depth} away from m_i. However, we found that the data dependency edges in G_i can greatly reduce the distance between these points in wild code. We also found that increasing n_{iter} improves performance up to n_{depth} (the diameter of the network). To find the optimal hyperparemeters for our CVE dataset, we used Bayesian parameter optimization [1]. For each trial, the optimiser trained a new model on a new dataset given the selected n_{depth}, n_{iter}, and other DNN parameters such as depth and learning rate. In our case, the optimizer found $n_{iter} = n_{depth} = 50$ to be the optimal setting. In general, the issue of scoping a program slice is an open research problem, and slicing is currently the best way to extract concise samples from G.

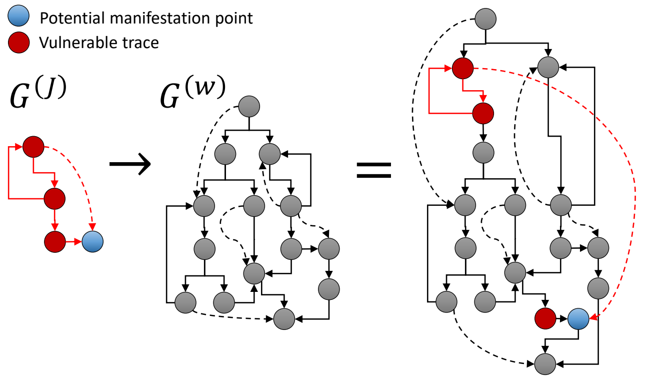


Figure 3: An illustration of an ePDG from the wild $G^{(w)}$ being augmented with a synthetic vulnerability trace from Juliet $G_i^{(J)}$.

4 Data Augmentation

Motivation. To create a model that can operate on real projects, it must be trained on samples that reflect the real world, having large bodies of irrelevant code and benign patterns between root causes and manifestation points. Some line-labeled datasets exist, such as the Juliet C/C++ Test Suite from NIST [26]. However, they consist of short synthetic programs that do not reflect real-world code. In the absence of realistic line-labelled datasets, we propose a data augmentation technique that combines vulnerabilities from synthetic line-labeled datasets with real-world code to generate realistic training samples.

Method. To accomplish this, we augment a 'clean' ePDG from a real-world project[8] from GitHub by splicing into it multiple Juliet ePDG subgraphs containing vulnerabilities. This generates a training sample with many positive manifestation points, each of which are extracted separately during program slicing.

Formally, given an ePDG of a real-world project $G^{(w)}$ and a set of vulnerable subgraphs extracted from the Juliet dataset $G_i^{(J)} \in J$, we augment $G^{(w)}$ as follows (illustrated in Figure 3):

Augmentation Procedure

```
1. Select a $G_i^{(J)}$ at random from $J$
2. Choose a random path of length $n$, where $3 \leq n \leq n_{depth}$
3. Select a random path $p$ in the control-flow of $G^{(w)}$
   with length $n$
4. If all nodes in $p$ are unmarked AND $p$ is at least $s$ hops
   away from other marked nodes; Then
    (a) Split $G_i^{(J)}$ in the middle of its control-flow
    (b) Insert and connect the first half at $p[0]$ and the
        second half at $p[n]$
    (c) Mark all nodes along $p$
5. Else; count = count + 1 //There was a collision
6. If count > limit; Then return $G^{(w)}$
7. Else; loop to Step 1
```

When VulChecker trains on the augmented $G^{(w)}$, it will cut a sample $G_i^{(w)}$ for every potential manifestation node, including the positive ones injected from Juliet (blue in the figure). However, since we place samples S steps away from nodes marked in red, $G_i^{(w)}$ may sometimes overlap with $G_j^{(w)}$ for $i \neq j$. The reason for this is to help the model learn to focus on the target terminating manifestation point.

Validity. Since our augmentation process splices multiple ePDGs, it may produce samples where a vulnerability ePDG subgraph lies on an infeasible path (i.e., it is dynamically unreachable) in the augmented ePDG. As is typical of static analysis tools, VulChecker considers both feasible and infeasible paths in a program. Therefore, it can still learn from such samples provided (1) the augmentation maintains the vulnerabilities' data-flow and static reachability properties, and (2) splicing ePDGs does not otherwise invalidate inserted vulnerabilities. To ensure (1), our augmentation process preserves the data-flow and static reachability of $G^{(w)}$ and $G^{(j)}$. This is illustrated in Figure 3 where (A) the augmented ePDG still contains the same data-flow edges found in $G^{(j)}$ (the dashed lines) and (B) all nodes remain statically reachable in the same order. Regarding (2), augmentation cannot invalidate a vulnerability because nodes in $G^{(j)}$ and $G^{(w)}$ are guaranteed not to interfere with each other. Non-interference among data-flows follows from our derivation of ePDGs from SSA (static single assignment) form LLVM IR. In SSA form, all data values are defined exactly once, meaning no node from $G^{(w)}$ can redefine a data value created by a node in $G^{(j)}$. Non-interference among static control-flows arises from our augmentation procedure that ensures that the second half of the vulnerability subgraph remains statically reachable from the first half via at least one program path.[9]

In summary, the augmented samples $G_i^{(w)}$ help the model generalize to more realistic scenarios because (1) they contain real-world code and (2) they teach the model to search through greater distances and noise to find the root cause. We note that the same augmentation approach can be extended to other code representations such as CFG, PDG and CPG.

5 Evaluation

In this section, we evaluate VulChecker's performance as a tool for detecting and localizing vulnerabilities in source code. We accomplish this by evaluating VulChecker's performance at detecting vulnerabilities in the wild (real CVEs) when only trained on augmented (synthetic) datasets. We also explore the impact of the augmentation as well as the tool's precision to understand its practicality.

5.1 Experiment Setup

In this section we present our experiment setup. We note that all of our code has been published online, including our datasets and trained models, for use by the software and research communities.

5.1.1 Experiments

To evaluate VulChecker, we performed 3 experiments:

EXP1: Performance. To measure VulChecker's general performance, we use the area under the curve (AUC) measure where the values of 1.0 and 0.5 indicate a perfect classifier and random guessing respectively. For application performance (where a detection threshold is set), we measure the false positive rate (FPR), true positive rate (TPR), and accuracy (ACC).

As a baseline, we compare VulChecker to five different source code vulnerability detection tools and methods (two which detect vulnerabilities in code regions and three which perform line-level localization):

Region-level detectors. The first is the approach of [31] where ASTs of functions are converted into sequences of nodes. Unknown code is then assigned a vulnerability score based on its longest common subsequence (LCS) similarity to samples of vulnerable code. We contacted the authors for their source code, but did not hear back from them. Therefore, we implemented their solution to the best of our ability. We normalized the LCS values between 0 and 1 based on the length of the largest sequence being compared. We note that although their work performs function-level localization, it can be used to classify the CWE. Therefore, we use this work as a baseline because it can be compared directly to VulChecker's detection performance. The second is Gemini, a deep learning-based binary level code similarity tool described in section 2.

[8]We ensure these projects are free of known CVEs (see section 3.2.3).

[9]We manually verified that our augmentation process was sound with respect to these two properties across a random sample of approximately 100 augmented ePDGS.

Line-level detectors. The first is Helix QAC, a licensed commercial SAST tool by Perforce.[10] QAC is a static code analyzer that automatically scans code for CWE violations (based on C/C++ coding rules). It both localizes at the line-level and classifies vulnerabilities like VulChecker. The second is an open source SAST tool called Cppcheck[11] which also performs localization and classification. The third is VulDeeLocator (described in section 2). In the original paper, VulDeeLocator was trained on a mix of CWEs, meaning that users could not tell which CWE was predicted (only that a given line is vulnerable). In our implementation of VulDeeLocator, we mirror VulChecker and train separate model on each CWE to refine the predictions. We used the author's code[12] but found it very limited: (1) it cannot parse lines which will result in large slices and (2) it cannot efficiently parse large projects. Therefore, we modified their code to reduce these limitations to the best of our ability.[13] We ensured that all positive cases were included although many of negative cases were omitted. Note, this may decrease VulDeeLocator's false positive rates in EXP1.

We selected the Cppcheck and QAC because they are well-known SAST tools and they perform both line-level localization and vulnerability classification like VulChecker. We selected [31] (LCS) because it captures the performance of both AST and pattern matching-based approaches. We selected Gemini because it has some parallels to VulChecker: it uses the Structure2Vec and it detects vulnerabilities on code structure (CFGs). However, it differs from VulChecker in that it only operates on binary, does not use ePDGs, requires a large labeled dataset, and it cannot localize vulnerabilities to the line-level (only function-level). Finally, we use VulDeepLocator since it uses deep learning to perform line level localization like VulChecker.

EXP2: Precision. To support claims that VulChecker is a practical and helpful tool for developers, we employed an experienced vulnerability analyst to review the top 100 results, for each CWE and evaluated system, on a hold out set and measured precision. Examining thousands of cases[14] to determine whether they are buggy (and exploitable) is challenging for large real-world projects due to hard problems like determining reachability (i.e., determining whether the conditions to trigger a bug can be satisfied starting from the program's entry point). Since it is infeasible to confirm exploitability for so many cases, we employed the following methodology to determine the precision of the system:

1. The analyst first examined the source code for each case and determined whether the expected behavior for the predicted CWE is present and whether the ePDG contains any checks to handle that CWE. For example, a positive use-after-free case must contain a free statement, followed by a use of the freed variable name, without any checks in-between for whether the variable has been freed or not. A positive integer overflow case must contain arithmetic statements followed by a use of the calculated result, without checking for integer wraparound. The result of this step is two buckets: cases that cannot contain the predicted CWE and ones that might.
2. Next, the analyst examined every known CVE for the projects used in this experiment and collected their corresponding patches. With this information, the analyst examined each case in the "maybe buggy" bucket and identified the ones that match already known CVEs. The result is a bucket of detected, previously known, vulnerabilities.
3. Finally, for the cases that might be buggy and could not be matched to a known vulnerability, the analyst took a random sample and attempted to verify their exploitability using industry best-practices like fuzz testing and manual reverse engineering. This process took several weeks.

For the purpose of this experiment, we consider cases in the not buggy bucket from Step 1 to be false positives and cases in the latter buckets (maybe buggy, matched to prior CVE, verified novel vulnerability) to be true positives. This decision is based on the fact that a "maybe buggy" case can still be a vulnerability, even if the analyst failed to verify it given time constraints.

For the purposes of ethical disclosure, we chose to disclose novel bugs that were *verified exploitable*.[15] We also provide a video demo of VulChecker operating as plugin for Visual Studio.[16]

EXP3: Ablation. In our paper, we claim that training on augmented data is beneficial because it is challenging to obtain real world instruction or line-level labeled datasets. However, it is not clear how much the augmentation process helps the overall performance. To demonstrate the contribution of the augmentation process, we perform an ablation study by contrasting a model trained on augmented data to one trained on synthetic data.

Case Studies. By using VulChecker on our datasets, we identified a zero-day vulnerability and incorrect CVE information. Details on these cases can be found in the appendix.

5.1.2 Datasets

We collected a wide variety of C/C++ project for training and testing VulChecker and the baselines. A description of these projects, including a mapping between the projects and the experiments, can be found in the appendix in Table 6.

Train set. To train the baselines, we used the Juliet C/C++ dataset denoted D^{c}_{jul} for CWE c. The dataset has the following number of samples for each CWE (D^{190}_{jul}: *3960*, D^{121}_{jul}: *4944*, D^{122}_{jul}: *5922*, D^{415}_{jul}: *960*, D^{416}_{jul}: *459*).

[10] https://www.perforce.com/products/helix-qac
[11] https://github.com/danmar/cppcheck
[12] Original implementation: https://github.com/VulDeeLocator
[13] Our implementation: https://github.com/evandowning/VulDeeLocator
[14] Top 100 cases, for each CWE, for each evaluated system.

[15] The authors feel that reporting cases that might not be true bugs, without a reproducible proof-of-compromise (PoC), would be disrespectful of the developer's time.
[16] https://tinyurl.com/mucpt67x

To train VulChecker, we used D_{jul} to augment 20 'clean' projects collected from GitHub: A 'clean' project was selected if it (1) did not contain a CVE for the given CWE, (2) had at least 40k lines of source code, and (3) was a `cmake` project.[17] Then, for each CWE, we augmented these projects using D_{jul}. We denote this augmented training set as D^{c}_{aug} for CWE c. Overall, there were approximately 6k–24k positive manifestation points and 2 million negative points per CWE (with some projects having over 200 million nodes in total). To handle class imbalance, we down sampled the negative points to equal proportions.

We note that data the augmentation process is part of VulChecker's algorithm (it cannot be applied to the other baseline models). However, VulChecker and the baselines were all trained on the same positive data since D_{jul} is captured in D_{aug}.

Test sets. In our evaluations, we used open source projects collected from the Internet. To evaluate the models in **EXP1**, we collected projects from GitHub which contain CVEs: First, we collected a list of all relevant CVEs using the NVD database and filtered out all CVEs that were not tagged with one of our CWEs (leaving 3,788). For each CWE, we then filtered out all of those without version information and that were confirmed to be closed source (leaving 524). We then manually collected all of the projects, filtered out any project that does not use the `cmake` compilation system, and verified the CWE label in the code. In the end, we had 19 projects with approximately 35 CVEs. Of these CVEs, 14 were from 2019 and 2020 and the majority had CVSS ranks (severity levels) of medium or high. We denote this dataset as D^{c}_{cve} for CWE c. For each CVE, a vulnerability researcher tagged the positive manifestation point in the source code (all other points are considered negative). In total, we used the following number of projects containing one or more relevant CVEs (D^{190}_{cve}: 9, D^{121}_{cve}: 2, D^{122}_{cve}: 4, D^{415}_{cve}: 2, D^{416}_{cve}: 7). These projects contained millions of nodes. For more information on these projects, see Table 5 in the appendix.

To evaluate the precision of VulChecker in **EXP2**, used 9 more *unlabeled* projects collected from GitHub. These projects were selected if they were `cmake` and had no reported CVE. We denote this dataset as D_{out}.

Finally, in **EXP3** (ablation on data augmentation), we used a CWE190 dataset consisting of holdout data from D_{jul}, 1 project from D^{190}_{cve} and 5 more 'clean' wild projects from GitHub.

5.1.3 Model Configuration

We configured all of the CWE models with the same hyper-parameters based on results from experimentation and Bayesian optimization [1]. For program slicing, we used a backwards cut depth of $n_{depth} = 50$. For M_G, we used an embedding size of 32, 9 layers in the network σ, and performed $n_{iter} = 50$ propagation iterations. For M_C, we used 7 dense layers of 32 neurons each. The entire model was implemented using PyTorch and trained end-to-end with a learning rate of 0.0001 for 100 epochs on a NVIDIA Titan RTX GPU with 24GB of RAM.

[17]Our current implementation of the LLVM ePDG extractor supports `cmake`, but the pipeline is not fundamentally restricted to a particular build system.

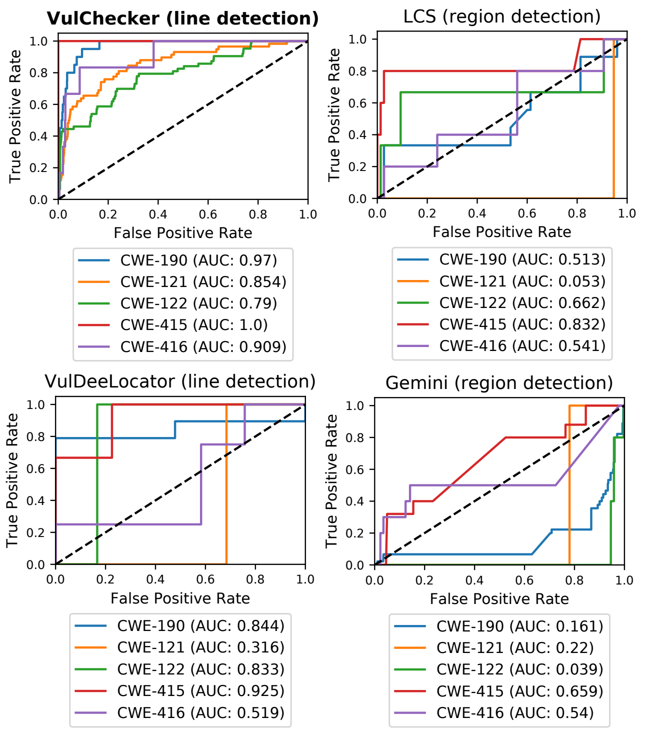


Figure 4: Performance of the line-based detectors (left) and region-based detectors (right) in detecting CVEs in the wild.

The models in EXP1 were setup as follows. For LCS, we followed the approach of the authors [31] and we compared all 24k functions in Juliet to a random subset of 400 functions from the wild,[18] ensuring that all 23 vulnerable functions were included as well. For Gemini and VulDeeLocator, a separate model was trained on each CWE in the Juliet Dataset (i.e., D^{c}_{jul}). For VulChecker, the models were trained on the respective augmented CWE datasets D^{c}_{aug}. Finally, for QAC and Cppcheck, the default configurations were used.

5.2 EXP1: Performance

In Figure 4 we present the results of the baselines which are open-source on D_{cve}. In the figure, we plot each of the models' receiver operating characteristic (ROC) curves and provide the models' AUCs. The ROC plot shows the tradeoff between the FPR and TPR at every threshold. In particular, we are interested in the left side of the plot where the model can be tuned to achieve a low FPR. With a FPR of 0.01, we found that VulChecker was able to accurately identify 35% of all 63 positive manifestations in CWE-122, 45% of the 20 manifestations in CWE-190, and all 3 manifestations in CWE-415. When relaxing the threshold to a FPR of 0.1, VulChecker can identify 95%, 64%, 46%, 100%, and 83% of the CVE manifestations for CWEs 190, 121, 122, 415, and 416, respectively. A list of the 24 CVEs detected at this rate can be found in the Appendix along with the performance at

[18]400 random negative samples are taken due to the slow speed of LCS.

Table 3: Baseline comparison against a commercial SAST tool in detecting CVEs in the wild.

	VulChecker @ FPR 0.05			VulChecker @ FPR 0.1			Helix QAC		
	Lines		CVEs	Lines		CVEs	Lines		CVEs
CWE	**TP**	**FP**	**TP**	**TP**	**FP**	**TP**	**TP**	**FP**	**TP**
190	9	55	3	12	112	6	1	2	1
121	7	33	7	9	112	9	4	230	1
122	1	6	1	1	6	1	4	241	1
415	3	0	2	3	0	2	0	5	0
416	4	6	4	6	228	6	0	0	1
Total	**24**	**100**	**17**	**31**	**458**	**24**	**9**	**478**	**4**

different set FPR rates (Table 4). Cppcheck is not included in the figure because it did not have any TPs. Instead, Cppcheck produced 22 FPs for CWE-190 and 1 FP for CWE-415.

The results in Figure 4 show that VulChecker outperforms the other methods by a large margin. This is because AST (LCS), CFG (Gemini) and linear (VulDeeLocator –see section 2.3) structures cannot capture all code behaviors, such as data dependencies, whereas VulChecker's ePDGs captures these behaviors explicitly.

In Table 3 we compare the performance of VulChecker to the closed-source solution (Helix QAC) on D_{cve} (TP and FP are the true positive and false positive counts respectively). When setting VulChecker's FPR to be on par with the commercial tool QAC (FPR 0.1), we find that VulChecker detects significantly more CVEs than QAC in D_{cve} (24 vs. 4). Unlike QAC, which cannot adjust its sensitivity, VulChecker can be adjusted. Specifically, we can raise the FPR to 0.2 and detect 31 CVEs (with double the FPs) or lower the FPR to 0.05 and detect 17 CVEs (with a quarter the FPs). We note that although VulChecker and QAC have ∼460 FPs at the line-level, there are hundreds of thousands of lines of code in the target projects. Therefore, 460 FPs is reasonable.

We note that since the software projects in D^c_{cve} are older versions of the software (Table 5), they contain a number of bugs and vulnerabilities that we have labeled as negative. Therefore, the FPR is actually lower since the top results contain other bugs and vulnerabilities. Through a manual inspection of the top 50 results for each CWE, we found that 3–7 additional instances of each CWE were detected.

In summary, VulChecker demonstrates good performance in detecting the exact lines and instructions of exploitable vulnerabilities (CVEs) in real world projects. Moreover, it can operate with the same FPR as a commercial SAST tool while detecting x6 more CVEs. This is a meaningful result, since the model was trained for 'free' on augmented data labeled with only synthetic samples.

5.3 EXP2: Model Precision

In deployment, a user would train VulChecker on both D^c_{aug} and D^c_{cve} to obtain a more complete and accurate model. To evaluate the precision and practicality of VulChecker in this scenario, we manually inserted different instances of vulnerability CWE-190 into `libgd`. We found that when a model is trained on both D^{190}_{aug} and D^{190}_{cve} it increases the rank of these manifestation points from the 20th–64th positions to the 1st position (highest score). Ranking is achieved by sorting results according to model's softmax confidence scores.

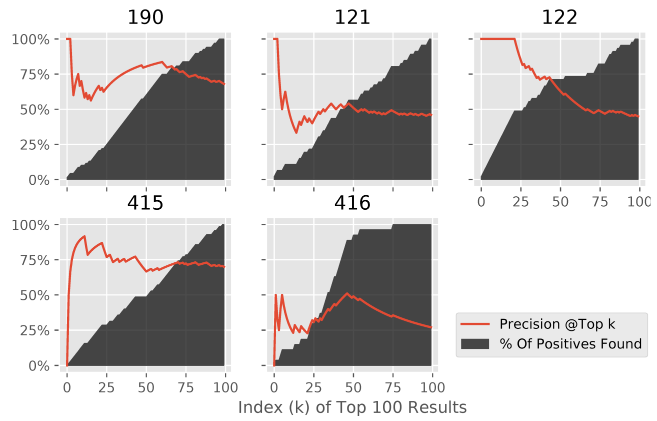


Figure 5: The precision (hit rate) of VulChecker on the holdout set when trained on both augmented and CVE datasets.

With this knowledge, we trained new models on both D^c_{aug} and D^c_{cve} and executed each of them on our holdout set D_{out}. Since the holdout set has no ground truth, we hired a vulnerability analyst to examine the top 100 results (from 1 million potential manifestation points). Figure 5 plots the precision of VulChecker on the top k results for each CWE. A rise in precision indicates sequence (cluster) of positive cases found. As expected, the precision drops as k increases since the model's confidence lowers over k. Using the analyst's findings, we found that 50-80% of the top 50 results, and 45-70% of the top 100 results were true positives, according to our criteria defined in Subsection 5.1. Concretely, the following counts of true positive cases were found in the top 100 results for each CWE (CWE190:68, CWE121:47, CWE122:46, CWE415:71, CWE416:28).

The analyst then took these true positives and attempted to manually match them with previously known CVEs, using their patches as reference. For CWEs 190, 121, 122, 415 and 416 we found 23.5%, 32.6%, 89.0%, 24.2% and 14.8% of the true positives to be 17 verified CVEs. Note that because a CVE can match multiple related lines in the source code, multiple true positives cases can correspond to the same CVE. The CVEs identified are included in Appendix A.

Lastly, the analyst attempted to verify the exploitability of the remaining true positive cases using standard practices like fuzz testing and manual reverse engineering. This yielded 1 verified zero-day vulnerability, which we disclosed to developers for patching. A case study of this vulnerability is included in Appendix B.

In summary, VulChecker provides valuable information to developers with a minimal number of false alarms. This means that in deployment, VulChecker can be a practical tool for identifying vulnerabilities in source code.

5.4 EXP3: Ablation Study on Augmentation

In Figure 6, we present the ROC curves and AUC values for a model trained on only synthetic data. The model performs extremely well on other synthetic samples because the samples are short and have virtually no noise. However, the same model fails to generalise to the real world software and vulnerabilities found in D_{cve}. This is a troubling insight since many prior proposals in this domain have evaluated their model's performance on the synthetic datasets like SARD [14, 19, 20, 22, 30, 35, 38], so it is not clear whether these can perform well in practice. We note that the proposed augmentation process can be applied to these works since the algorithm is agnostic to CFG, PDG, and CPG code graphs. However, we leave this comparative study for future work since it is outside the scope of our target application (line/instruction level vulnerability detection).

When contrasting the results in Figure 6 to those of Figure 4, we can clearly see the benefit and importance of the data augmentation. This is also apparent in the knowledge captured by each of the models: In Figure 7, we plot M_G's embeddings of G_i' (prior to the DNN classification) when trained on synthetic CWE-121 samples (left) and augmented CWE-121 samples (right). From the plots, we can see that the model trained on synthetic data learns a very simple representation since it is easy for it to separate negative and positive manifestation points. On the other hand, the vulnerabilities in augmented data have much longer distances between the root cause and manifestation points. This forces the model to look deeper into G_i' and to learn how to distinguish benign code from vulnerable code while overlooking irrelevant code. This can be seen in Figure 7 where the model trained on augmented data has difficulty separating the concepts in the ground truth (top right), but succeeds in associating the learnt concepts in samples from the wild (bottom right). In contrast, the model trained on synthetic data easily separates the concepts in training (top left) but fails to associate/identify any of the samples from the wild (bottom left).

In summary, the proposed data augmentation strategy is a cheap and effective way to boost a model's performance on real software projects. Given the shortage of fine-grained labeled datasets from the wild, this augmentation approach enables the research, development, and actual deployment of line-level classifiers.

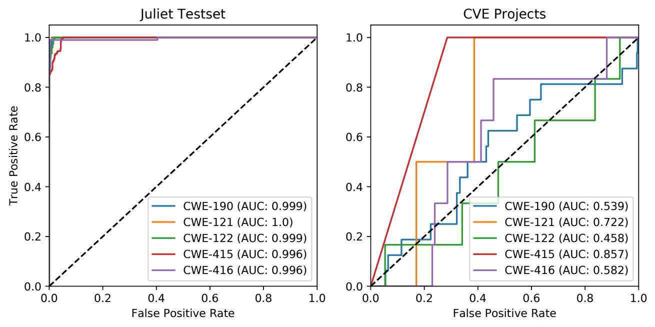


Figure 6: Performance of VulChecker when trained on synthetic data, then either tested on synthetic (left) or tested on real data (right).

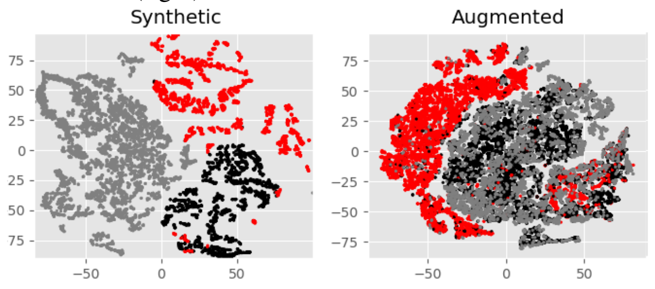


Figure 7: The embeddings of G_i' learnt by S2V when trained on synthetic (left) or augmented (right) data. Black (negative) and red (positive/vulnerable) points are samples taken from the model's training set. Grey points are embeddings of novel samples taken from the wild. For visualization, all plots have been reduced from 32 to 2 dimensions using T-SNE.

6 Assumptions & Limitations

In this framework we made several assumptions about the environment and use cases.

Responsiveness. In order to lower the code into LLVM IR, the code must be able to compile. This means that new reports to the developer will only become available at certain time-frames and not in real-time while typing. For example, when the developer completes a line of code or entire segment. We believe this is a reasonable delay since the programmer will still be engaged in the code and can quickly resolve the problem before moving on.

Security. An advanced attacker may use adversarial machine learning [10] to poison the training set (e.g., include crafted negative samples which will cause the model to miss certain vulnerability patterns in deployment). Although this attack is possible, it is challenging because the attacker must know which GitHub projects will be collected.

Regarding attacks on a trained model, it may be possible to generate adversarial perturbations in S such that G_i will not be detected by M as vulnerable. This is less of a threat in private projects with trusted contributors (e.g., a software company). Regardless, there are some significant challenges that the attacker must accomplish to generate perturbations: (1) it is an open research question whether it is possible to generate adversarial perturbations on source code that will compile, and (2) the mapping of S to G is not differentiable so the attacker can only operate on G_i'.

Integrity. Since we use projects from the wild for our augmentation, it is possible that vulnerabilities in these projects exist and will be considered as negative during training. This may confuse the model and impact the performance. Although we showed that this technique works well empirically, further research is needed to understand how the quantity and quality of these bugs impact a trained model.

Depth. A fundamental limitation of VulChecker is the range of parameter n_{depth}. If n_{depth} is substantially large then it would become prohibitively expensive to train and execute VulChecker on every potential manifestation point. Moreover, the size of G can negatively impact the models ability to identify long-distance relationships [41]. Therefore, when a root cause is significantly far from a positive manifestation point, VulChecker will not be able to confidently detect the bug. We note that ePDGs consider data dependency edges which significantly shorten the distance between root causes and manifestation points in G. However, the problem of efficient long-distance causality in static analysis is an open problem, unresolved in existing approaches (see section 2).

7 Conclusion

There is a large gap between the current state-of-the-art and a practical deep learning SAST tool that can be used by developers. In this paper, we proposed VulChecker, the first tool that can both perform line/instruction level vulnerability detection and classify the indicated vulnerability. We have also proposed (1) a new code representation (ePDG) for low level GNN tasks on software and (2) a novel data augmentation strategy to help GNN-based SAST tools work on real world software.

Ethical Disclosures

All previously unknown vulnerabilities found by VulChecker in this study have been disclosed to the respective software developers for remediation.

Acknowledgment

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract HR00112090031. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA. This material is also based upon work supported by the Zuckerman STEM Leadership Program.

References

[1] "scikit-optimize: Sequential model-based optimization in python." [Online]. Available: https://scikit-optimize.github.io/stable/

[2] "Application security | cyberres," https://www.microfocus.com/en-us/cyberres/application-security, 2021, (Accessed on 07/23/2021).

[3] "Clang static analyzer," https://clang-analyzer.llvm.org/, 2021, (Accessed on 07/23/2021).

[4] "coverity - google search," https://www.google.com/search?q=coverity&oq=coverity&aqs=chrome..69i57j0l9.1102j0j4&sourceid=chrome&ie=UTF-8, 2021, (Accessed on 07/23/2021).

[5] "Flawfinder home page," https://dwheeler.com/flawfinder/, 2021, (Accessed on 07/23/2021).

[6] "Helix qac for c and c++ | perforce," https://www.perforce.com/products/helix-qac, 2021, (Accessed on 07/23/2021).

[7] "Infer static analyzer | infer | infer," https://fbinfer.com/, 2021, (Accessed on 07/23/2021).

[8] S. Arakelyan, S. Arasteh, C. Hauser, E. Kline, and A. Galstyan, "Bin2vec: learning representations of binary executable programs for security tasks," *Cybersecurity*, vol. 4, no. 1, pp. 1–14, 2021.

[9] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, p. 106576, 2021.

[10] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay, "Adversarial attacks and defences: A survey," *arXiv preprint arXiv:1810.00069*, 2018.

[11] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[12] A. Chaudhary, H. Mittal, and A. Arora, "Anomaly detection using graph neural networks," in *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. IEEE, 2019, pp. 346–350.

[13] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[14] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2019, pp. 41–50.

[15] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2702–2711. [Online]. Available: http://proceedings.mlr.press/v48/daib16.html

[16] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489.

[17] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Network and Distributed System Security Symposium*, 2020.

[18] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[19] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, "Automated vulnerability detection in source code using minimum intermediate representation learning," *Applied Sciences*, vol. 10, no. 5, p. 1692, 2020.

[20] X. Li, L. Wang, Y. Xin, Y. Yang, Q. Tang, and Y. Chen, "Automated software vulnerability detection based on hybrid neural network," *Applied Sciences*, vol. 11, no. 7, p. 3201, 2021.

[21] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[22] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[23] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[24] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[25] MITRE, "Cve," https://cve.mitre.org/, 2021, (Accessed on 07/23/2021).

[26] NIST, "Software assurance reference dataset," https://samate.nist.gov/SARD/, (Accessed on 07/23/2021).

[27] D. Rolnick, A. Veit, S. Belongie, and N. Shavit, "Deep learning is robust to massive label noise," *arXiv preprint arXiv:1705.10694*, 2017.

[28] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[29] T. Spring, "Microsoft quietly patches another critical malware protection engine flaw | threatpost," https://threatpost.com/microsoft-quietly-patches-another-critical-malware-\protect\@normalcr\relaxprotection-engine-flaw/125951/, 2021, (Accessed on 07/23/2021).

[30] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Learning to map source code to software vulnerability using code-as-a-graph," *arXiv preprint arXiv:2006.08614*, 2020.

[31] N. Visalli, L. Deng, A. Al-Suwaida, Z. Brown, M. Joshi, and B. Wei, "Towards automated security vulnerability and software defect localization," in *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, 2019, pp. 90–93.

[32] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2020.

[33] T. Wang, C. Song, and W. Lee, "Diagnosis and emergency patch generation for integer overflow exploits," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2014, pp. 255–275.

[34] D. R. White and S. P. Borgatti, "Betweenness centrality measures for directed graphs," *Social networks*, vol. 16, no. 4, pp. 335–346, 1994.

[35] Y. Wu, J. Lu, Y. Zhang, and S. Jin, "Vulnerability detection in c/c++ source code with graph representation learning," in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2021, pp. 1519–1524.

[36] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 363–376. [Online]. Available: https://doi.org/10.1145/3133956.3134018

[37] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[38] M. Zagane, M. K. Abdi, and M. Alenezi, “Deep learning for software vulnerabilities detection using code metrics,” *IEEE Access*, vol. 8, pp. 74 562–74 570, 2020.

[39] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” *arXiv preprint arXiv:1909.03496*, 2019.

[40] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, “μvuldeepecker: A deep learning-based system for multiclass vulnerability detection,” *IEEE Transactions on Dependable and Secure Computing*, 2019.

[41] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, “Layer-dependent importance sampling for training deep and large graph convolutional networks,” *Advances in neural information processing systems*, vol. 32, 2019.

Appendix

A Additional Results on CVE Detection

The following breaks down which CVEs are detected by VulChecker and QAC for different FPRs:

- With an FPR set to 0.05, and trained on augmented data only, VulChecker successfully identified the following CVEs in the projects listed in Table 5: CVE-2011-0904, CVE-2016-2799, CVE-2016-5824, CVE-2016-9584, CVE-2016-9591, CVE-2017-12982, CVE-2017-14520, CVE-2017-15565, CVE-2017-15642, CVE-2018-10887, CVE-2019-10020, CVE-2019-10024, CVE-2019-13590, CVE-2019-14288, CVE-2019-17546, CVE-2019-8356, CVE-2020-15305, and CVE-2020-15389.
- With the FPR set to 0.1 we also detect CVE-2014-6053, CVE-2016-2799, CVE-2017-15372, CVE-2017-9775, CVE-2019-3822, and CVE-2019-5435.
- With the FPR set to 0.2 we also detect CVE-2017-8816, CVE-2018-14618, CVE-2018-20330, CVE-2018-7225, CVE-2019-14289, CVE-2019-5482, and CVE-2020-27828.
- Perforce QAC detects: CVE-2019-3822, CVE-2018-20330, CVE-2014-9655, and CVE-2020-15305.
- The 17 CVEs detected in EXP2: CVE-2005-3628, CVE-2008-3522, CVE-2009-3605, CVE-2013-1788, CVE-2016-10251, CVE-2016-8693, CVE-2016-8886, CVE-2016-9396, CVE-2017-7698, CVE-2018-5727, CVE-2018-5785, CVE-2019-13282, CVE-2019-13289, CVE-2019-16927, CVE-2019-5435, CVE-2019-9200 and CVE-2022-27337.

B Case Studies

Zero-day detection. During evaluation, VulChecker uncovered a novel and exploitable zero-day vulnerability in one of the analyzed C++ projects, demonstrating its ability to identify new bug patterns. The vulnerable code snippet is shown in Figure 8 of the appendix, with some lines removed for brevity. In this case, the vulnerable function is is a lexical parser designed to process PDF files. Unfortunately, such parsing is difficult to implement correctly, and the function is almost 100 lines long, posing a challenge for both manual and SAST-assisted code review. As it turns out, due to how the developers nested calls to `hd_read_byte`, with only the outer-most code loop checking the buffer's bounds, a maliciously crafted PDF file can cause the final `default` switch case to write outside the buffer, causing a heap overflow.

Fortunately, VulChecker is able to detect the overflowing instruction in a matter of minutes as a manifestation point for CWE-122, which we verified with an expertly crafted proof-of-compromise (PoC). We have disclosed this vulnerability to the project's developers, who at the time of writing have acknowledged the issue and are developing a patch.

Supply Chain Risk Reduction. One detection made by VulChecker that surprised us occurred in Poppler version 0.10.6. In this case, VulChecker labeled a buggy line that we determined to be CVE-2009-0756, however this conclusion was initially perplexing because according to the official CVE advisory, the bug was only known to exist in versions 0.10.4 and earlier, not 0.10.6. However, once we downloaded the patch published by the developers and compared it to our version of Poppler, we discovered that we indeed had a vulnerable version of the library, despite having downloaded it directly from the vendor's website. This is significant because developers frequently check the library dependencies of their projects against CVE advisories for potential risks. In this case, without VulChecker's analysis, such checks would wrongly conclude that there is no need to worry about CVE-2009-0756 since the library's version falls outside of known vulnerable releases. In reality, the CVE *was* present in our experiment.

Table 4: Performance at Different FPR Rates

		CWE				
@FPR=		**190**	**121**	**122**	**415**	**416**
0.01	**TPR**	0.450	0.121	0.349	1.000	0.167
	FPR	0.005	0.006	0.009	0.000	0.005
	ACC	0.993	0.986	0.980	1.000	0.994
0.05	**TPR**	0.800	0.569	0.444	1.000	0.667
	FPR	0.036	0.049	0.019	0.000	0.029
	ACC	0.964	0.947	0.971	1.000	0.971
0.1	**TPR**	0.950	0.638	0.460	1.000	0.833
	FPR	0.095	0.091	0.062	0.000	0.086
	ACC	0.905	0.906	0.930	1.000	0.914
0.2	**TPR**	1.000	0.759	0.587	1.000	0.833
	FPR	0.165	0.185	0.156	0.000	0.086
	ACC	0.836	0.814	0.839	1.000	0.914

```
char *s = lb->scratch;
char *e = s + lb->size;
/* truncated for brevity */
while (1) {
    if (s == e) {
        s += pdf_lexbuf_grow(ctx, lb);
        e = lb->scratch + lb->size;
    }
    c = hd_read_byte(ctx, f);
    switch (c) {
        case EOF:
            goto end;
        case '(':
            /* truncated */
            break;
        case ')':
            /* truncated */
            break;
        case '\\':
            c = hd_read_byte(ctx, f);
            switch (c) { /* truncated */ }
            break;
        default:
            *s++ = c; /* BUG: overflow */
            break;
    }
}
```

Figure 8: Exploitable zero-day found by VulChecker. A lack of bounds checking while performing lexical parsing for PDF files can result in an overflow write.

C Details on the Datasets

Table 5: The CVE Projects used in the Testsets

CWE	Project	Version	$\lvert V \rvert$
190	curl curl	7.56.1, 7.64.1	573k, 584k
	libcurl curl	7.61.0, 7.63.0	571k
	libgit2	0.26.1, 0.27.2	18.2 mil., 18.7 mil.
	libjpeg turbo	2.0.1	274k
	libtiff	4.0.10	297k
	libvncserver LibVNCServer	0.9.11	197k
	sound.exchange sox	14.4.2	416k
121	poppler poppler	0.55	2.7 mil.
	sound.exchange sox	14.4.2	416k
122	curl curl	7.61.1	568k
	graphite2	1.3.5	228k
	jasper version	2.0.22	445k
	sound.exchange sox	14.4.2	416k
415	jasper version	2.0.11	417k
	openjpeg	2.3.1	256k
416	jasper version	2.0.11	417k
	libical	1.0.0, 2.0.0	1.1 mil., 973k
	openexr	2.5.1	890k
	openjpeg	2.3.1	256k
	sound.exchange sox	14.4.2	416k

*Reflects most recent listed version

Project		CWE Dataset 190	121	122	415	416	Category	Source Files	Lines of Code	GitHub Stars
avian	1.2.0	►	►	►	►	►	JVM	182	67897	1204
bdwgc-gc	7.6.0	►	►	►	►	►	Garbage Collector	114	35658	2021
Bento4	1.5.0, 1.5.1	►	►	►	►	►	Audio	308	54137	1361
curl	7.56.1, 7.64.1	●	○		○	○	Networking	788	171503	25236
	7.61.1			●						
	7.77.0	╬	╬	╬	╬	╬				
graphite2	1.3.5	○	○	●	○	○	Fonts	230	53219	117
hdContents		╬	╬	╬	╬	╬	PDF	32	8404	7
jasper	2.0.11	○	○		○	●	Images	237	61291	168
	2.0.22	►	►	●	►	►				
libarchive	3.2.0, 3.2.1	►	►	►	►	►	Compression	573	142818	1743
libcurl	7.61.0, 7.63.0	●	○		○	○	Networking	803	175816	25236
libiec61850	1.3.0	►	►	►	►	►	Networking	423	103682	485
libgd	2.2.3		○	○	○	○	Images	373	85035	732
libgit2	0.26.1, 0.27.2	●	○	○	○	○	Git	978	205102	8444
	1.1.0	╬	╬	╬	╬	╬				
libical	1.0.0, 2.0.0	○	○	○	○	●	Networking	287	87021	229
libjpeg-turbo	2.0.1	●╬	○╬	○╬	○╬	○╬	Images	284	79668	2882
libtiff	4.0.10	●	○	○	○	○	Images	279	103394	53
libvncserver	0.9.11, 0.9.12	●	○	○	○	○	Networking	138	41734	834
	0.9.13	╬	╬	╬	╬	╬				
libzip-rel	1.2.0	►	►	►	►	►	Compression	152	14364	530
libzmq	4.2.5		○	○	○	○	Networking	365	47324	7787
openexer	2.5.1	○	○	○	○	●	Images	816	282636	1224
openjpeg	2.1.2, 2.3.1	○	○	○	●	●	Images	559	226120	802
	2.3.1					●				
	2.4.0	╬	╬	╬	╬	╬				
poppler	0.55	►	●	►	●	►	PDF	685	218518	N/A
	0.10.6, 0.53	►	►	►	►	►				
sound_exchange	14.4.2	●	●	●	○	●	Audio	394	75803	N/A
tigervnc	1.7.1		○	○	○		Networking	419	45016	3360

○ Used in **EXP1** and **EXP2** to make D_{aug} for training (has no CVE for the CWE)
● Used in **EXP1** and **EXP2** to make D_{cve} for testing (has CVE label(s))
► Used in **EXP2** for testing to make D_{out} (no labels)
╬ Used in **EXP3** for testing and has no reported CVE for CWE190

Table 6: Details on the source code projects taken from the wild and used in this paper. Left: a mapping of which project was used for each dataset by CWE. Right: Statistics on the projects.