PAVUDI: Patch-based Vulnerability Discovery using Machine Learning

Tom Ganz
tom.ganz@sap.com
SAP SE
Germany

Erik Imgrund
erik.imgrund@sap.com
SAP SE
Germany

Martin Härterich
martin.haerterich@sap.com
SAP SE
Germany

Konrad Rieck
rieck@tu-berlin.com
Technische Universität Berlin
Germany

ABSTRACT

Machine learning has been increasingly adopted for automatic security vulnerability discovery in research and industry. The ability to automatically identify and prioritize bugs in patches is crucial to organizations seeking to defend against potential threats. Previous works, however only consider bug discovery on statement, function or file level. How one would apply them to patches in realistic scenarios remains unclear. This paper presents a novel deep learning-based approach leveraging an interprocedural patch graph representation and graph neural networks to analyze software patches for identifying and locating potential security vulnerabilities. We modify current state-of-the-art learning-based static analyzers to be applicable to patches and show that our patch-based vulnerability discovery method, a context and flow-sensitive learning-based model, has a more than 50% increased detection performance, is twice as robust against concept drift after model deployment and is particularly better suited for analyzing large patches. In comparison, other methods already lose their efficiency when a patch touches more than five methods.

CCS CONCEPTS

• **Security and privacy → Software and application security**;
• **Computing methodologies → Machine learning**.

KEYWORDS

Vulnerability Discovery, Program Representations, Patches

1 INTRODUCTION

A change to a program is typically accomplished through a patch, providing feature updates or fixes for bugs and vulnerabilities [47]. With the increasing adoption of continuous integration (CI) and continuous deployment (CD), the need to monitor and validate patches for potential bugs has become substantial [33]. To maintain a secure and reliable software development life-cycle, organizations must have a robust software quality assurance process in place to identify and mitigate security risks before a patch reaches production systems.

ACSAC 2023, December, 2023,
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The traditional approach to finding bugs in software relies on manual code reviews and extensive testing. However, this approach is time-consuming, resource-intensive, and prone to human error. Static program analysis, on the other hand, supports developers to identify potentially flawed code regions without actually running the program. Unfortunately, such static application security testings (SASTs) tools often report many false positive alerts, which consequently requires expensive manual triage. The problem arises from too general detection rules and the theoretical limits of static analysis including bug and vulnerability detection [27]. In practice, developing detection rules for SAST tools is an error-prone and tedious task [30]. Hand-crafted rules are often incomplete or too sensitive, resulting in unfavorable trade-offs between high false-positive and false-negative rates. The detection deteriorates even more when these rules are project-agnostic and intended to apply to a large number of applications.

As a remedy, methods for learning-based vulnerability detection have been proposed to automatically derive rules from historical data [30, 35, 58]. Current machine learning (ML) models have been shown to beat rule-based SAST tools with a much higher detection rate while pertaining to a lower false-positive rate [14, 43, 57, 58]. However, the prevailing ML models focus exclusively on features in local code regions, such as functions [58], statements [17] or slices [29]. Moreover, these models are not context- or flow-sensitive, and thus suffer from low generalizability and transferability in realistic settings [11, 12, 32, 39].

With patches, the situation is even worse, as there is often little time to test and validate them before they are released and it remains unclear how a security expert would apply learning-based SAST tools to commits, as a patch can potentially span over multiple disjoint modules, functions, and classes. Since patches are the only atomic unit defining the evolution of software, it makes sense to adapt existing methods or develop novel techniques that infer bugs in patches rather than identifying bugs on more fine-granular levels. Yin et al. [56] state that up to 24% of patches introduce new bugs, moreover, with open-source software, most patch developers are not even familiar with the entire code base.

The main research question addressed in this work is how to effectively identify and locate bugs in patches. As current learning-based analyzers do not consider patches, we can demonstrate that their detection performance is poor. Thus, we present a novel patch-based vulnerability discovery (PAVUDI) approach. We combine

graph neural networks with traditional taint analysis to identify and locate bugs in patches considering the entire application. We first formalize a new graph representation that allows security practitioners to analyze interprocedural data and control flow from potentially attacker-controlled sources to software-critical regions. Such a single graph captures the impact of a given patch on the system's security posture. Secondly, we design a learning model specifically for this graph representation to automatically infer vulnerable or flawed paths within a patch.

To empirically validate the effectiveness of our approach, we conduct a comprehensive evaluation of the proposed method on a dataset of security patches. We show that PAVUDI outperforms state-of-the-art methods under real-world conditions. Furthermore, we find that current approaches suffer from concept drift, that is, they lose their initial detection capabilities over time. PAVUDI is less affected by this drift, providing a more stable detection performance over time.

In summary, we make the following contributions in this work:

(1) *Interprocedural code representation.* We introduce and formalize a new graph representation of code for finding security vulnerabilities. The graph is context- and flow-aware and provides security-relevant information especially for bug discovery in patches.
(2) *Explainable graph learning model.* We implement a novel model architecture that is well suited for our graph representation and provides explainable results using causal structure learning.
(3) *Extensive empirical evaluation.* We compare PAVUDI against several different state-of-the-art vulnerability discovery models and implement several detection strategies to apply these models to patches.
(4) *Real-world findings.* We test PAVUDI in realistic scenarios by detecting previously unknown bugs in open-source software. More specifically, we apply our model to two popular C libraries and find five bugs in their most recent 100 commits.

The rest of this paper is structured as follows: We begin with an introduction to vulnerabilities and graph representations in Section 2, then we detail our problem setting in Section 3 and present our methodology in Section 4. In Section 6, we discuss our empirical evaluation and end with related work and conclusions in Section 7 and Section 8, respectively.

2 VULNERABILITIES IN PATCHES

Before presenting our approach to detect vulnerabilities in patches, let us first introduce the basic concepts of static non-learning and learning-based vulnerability discovery methods.

2.1 Vulnerability Discovery

To begin, we define the task of discovering vulnerabilities in a program. We aim to derive a single score that indicates the likelihood of a program being vulnerable based on a particular representation of it. This is expressed in Definition 1, which defines a decision function that takes a piece of code and maps it to the probability of it being vulnerable. Note, that this definition does not differentiate between x being a function, statement or patch.

Definition 1. *A method for static vulnerability discovery is a decision function $f : x \mapsto P(\text{vulnerable} \mid x)$ that maps a piece of code x to its probability of being vulnerable [20].*

Classic rule-based SAST tools can be described directly as a function f predicting vulnerabilities, by for instance, matching function calls against known patterns or applying *taint style analysis* by tracking the flow of user-provided values, checking buffer bounds, detecting undefined behavior and more. Learning-based methods for vulnerability discovery, on the other hand, build on a function $f = f_\theta$ parameterized by model weights θ that are obtained by training on a dataset of vulnerable and non-vulnerable code [23]. Compared to classic static analysis tools, learning-based approaches do not have a fixed rule set and thus can adapt to characteristics of different vulnerabilities in the training data. The primary differences among these approaches lie in the input program representation and the learning model, for instance, how the function depends on the model weights.

2.2 Vulnerabilities and Patches

There are several ways vulnerabilities can slip into program code during software development, ranging from a single patch to a series of complex and intertwined changes to a program. While there are approaches to trace a discovered bug back to the inducing changes, the other direction, namely identifying all commits sufficient for spotting an unknown vulnerability, is a hard problem in the general case. As a remedy, we focus in this work on vulnerabilities that are linked to one specific patch.

In particular, we consider a patch as vulnerability-inducing if its code changes either directly introduce the defect or are in close proximity to an existing one, so that vulnerable data passes through code changes as defined later in Definition 7. An example of such a patch is the heartbeat commit introducing a buffer-overread in CVE-2014-0160, as shown in Figure 1. Note that even though we restrict our scope to single vulnerability-inducing patches, their complexity can still be significant, covering dozens of disconnected regions across an entire code base.

2.3 Graph Representations

Since programs can be modeled as directed graphs [2, 6, 53], recent approaches make use of graph representations [14, 43, 58] for source code instead of flat token sequences [30, 35]. We refer to the resulting representation as a *code graph* and denote the underlying directed graphs as $G = G(V, E)$ with vertices V and edges $E \subseteq V \times V$. Moreover, the nodes and edges are attributed, that is, elements of V or E are assigned values in a feature space.

However, different code graphs capture different syntactic and semantic features. Recent works, for instance, rely only on syntactic features for neural code comprehension using the abstract syntax tree (AST) [2]. This is a tree representing the syntactic structure of source code.

Definition 2. *The abstract syntax tree (AST) of a function f is the result of parsing its source such that the leaf nodes in the resulting tree $G_A = G(V_A, E_A)$ are the literals and the edges E_A describe the composition of syntactic elements [1].*

The semantic attributes of a function can be captured in flow graphs for instance with the flow of control or the flow in information defined in Definition 3.

Definition 3. *The control flow graph (CFG) within a function f is $G_C = G(V_C, E_C)$ with the nodes $V_C \subset V_A$ being statements, and where the directed edges E_C describe the execution order of the statements $V_C \subset V_A$ [53]. The data flow graph (DFG) within a function f is $G_D = G(V_D, E_D)$ with the nodes $V_D \subset V_A$ being variable assignments and references, and where the directed edges E_D describe read or write access from or respectively to a variable [9].*

These graph representations allow us to reason about the order of the executed statements and the flow of information between variables. An analysis using these properties is considered flow-sensitive [32]. Finally, a call graph connects function call-sites with the function definitions as defined in Definition 4.

Definition 4. *The call graph (CG) within a program is defined as $G_{CG} = G(V_{CG}, E_{CG})$ where the nodes $V_{CG} \subset V_A$ being function call-sites and definitions, while the edges E_{CG} connect the caller with the respective function definition.*

An analysis using the call context of a program is considered context-sensitive [32]. Code graphs capture syntactic and semantic relationships between statements and expressions in programs. Based on these classical representations, combined graphs have been developed for vulnerability discovery. A popular one is the code property graph (CPG) by Yamaguchi et al. [53], which is a combination of the AST, CFG and program dependence graph. Other approaches use different combinations, for instance, combining the AST with the CFG and the DFG [9], called code composite graph (CCG) as defined in Definition 5.

Definition 5. *The CCG is a disconnected graph G_{CCG} for a program $P = \{f_1, f_2, .., f_n\}$ with $V = \bigcup_{i=1}^{n} V_A^i$ and $E = E_A \cup E_D \cup E_C$ combining the AST with the semantic information from the CFG and DFG.*

The components of a CCG are easily obtained during compilation, and capture syntactic features and information flow, which fits neatly into the definition of taint-style analysis, which we will revisit in Section 4 [54].

2.4 Graph Representation Learning

With the recent success of graph-based program representations, research has started to focus on graph convolutional networks (GCNs) [58]. These networks are a class of deep learning models realizing a function $f : G(V, E) \mapsto y \in \mathbb{R}^d$ that can be used for the classification of graph-structured data [36].

GCNs can be viewed as a generalization of convolutional neural networks (CNNs), just as an image can be viewed as a regular grid graph where each pixel denotes a node in the graph connected by edges to its neighboring pixels [51]. A graph convolutional network needs two mandatory input parameters, that is, an initial feature vector $X \in \mathbb{R}^{N \times F}$, with N being the number of nodes in the graph and F the number of features per node, and the topology commonly described by the adjacency matrix $A \in [0, 1]^{N \times N}$. The most popular GCN types belong to so-called message passing networks (MPNs) where the prediction function is computed by iteratively aggregating and updating information from neighboring nodes. One of the simplest MPNs is the one defined by Kipf and Welling [26]:

$$h^{(l)} = \sigma(\hat{A}\, h^{(l-1)} W^{(l-1)}) \tag{1}$$

with $h^0 = X$. Here, the intermediate representations are linearly projected and sum-wise aggregated according to the normalized adjacency matrix $\hat{A}$ with self-loops followed by a non-linear activation function. These GCNs can be stacked to learn filters with respect to larger neighborhoods. Other GCN layers use different aggregation and update mechanisms, for instance, instead of an multilayer perceptron (MLP), gated graph neural networks (GGNNs) use gated recurrent unit (GRU) cells to update the hidden state of nodes [28], while graph attention network (GAT) layers use attention mechanisms [42]. We refer the reader to the overview article by Wu et al. [51] for further details.

Because of the fitting premise of GCNs, they have been widely adopted for representation learning on code graphs. The graph-based approaches in recent literature outperform classical SAST tools and older sequential learning models such as VulDeepecker [30] or Draper [35]. Graph learning-based approaches like Devign [58] and ReVeal [11] can be considered state-of-the-art and provide strong results in their respective publications.

3 PROBLEM SETTING

Current research on learning-based static code analysis focuses on local code regions, for instance, functions [11, 58], slices [29], or small code gadgets [14]. However, software development revolves around changes that can span multiple files and functions. Concurrent versioning systems like Git allow software developers to track changes that may contain bug fixes or feature enhancements commonly denoted as *patch*:

Definition 6. *A patch (commit) $[P' \Rrightarrow P]$ is a transition from one program P' to another P. It consists of changed code lines and files commonly denoted as* hunk. *A patch is often associated with a Git commit and its unique identifier [47].*

Besides introducing new features or fixing bugs, a patch also potentially adds new bugs [56] which we want to detect. The decision function from Definition 1 does not specify how to apply existing methods to patches. A naive approach would be to glue together all snippets changed by a patch before applying a decision function that operates on function or statement level.

However, problems arise due to the difficulty in identifying and locating all possible changes within a commit that potentially introduces bugs. Consider the heartbleed bug (CVE-2014-0160) in the OpenSSL C library. The bug was introduced due to a feature change adding *TLS heartbeats* three years prior to the discovery of the actual security vulnerability. The commit[1] touches twelve different C files and five header files in two different packages. A static analyzer would need to check all changed functions in all changed files to find the defect causing the heartbleed vulnerability shown in Figure 1.

We can see that `memcpy` copies a buffer with the size obtained by the client. If the client proposes a buffer length larger than the target

[1] https://github.com/openssl/openssl/commit/481750

```
if (hbtype == TLS1_HB_REQUEST)
    {
    unsigned char *buffer, *bp;
    int r;

    /* Allocate memory for the response, size is 1 byte
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;

    /* Enter response type, length and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

Figure 1: Buffer over-read in dtls1_process_heartbeat().

buffer, it effectively triggers a heap-based buffer over-read. Finding this bug among all the changed files seems like finding the needle in the haystack. A static analyzer would need to identify `memcpy` as a critical code statement, consider `payload` user-controlled and detect that there is no sanitization in-between.

We conclude that locating bugs in patches naturally comes with several problems that need to be addressed:

(1) *Context-sensitive changes.* Patches may only touch certain functions and modules but need to be analyzed in the surrounding context. A bug typically spans over multiple modules [57], that may be associated with a patch but do not have to be directly affected by it.
(2) *Non-coherent changes.* A patch may not correspond to a single feature change but potentially touch multiple modules that do not necessarily have to be associated with each other. Applying a SAST on all changed components may significantly increase the reported false positive alerts.
(3) *Evolution of software.* The learning-based discovery of vulnerabilities has already been addressed in research [11, 47, 58], however, finding a patch that introduces a bug is non-trivial, as a program may undergo several changes before a bug actually manifests. In addition, the feature representation of the program may change over time, causing the performance of a learning-based analyzer to degrade over time as well.

4 METHODOLOGY

PAVUDI is inspired by classic taint analysis, a dynamic program analysis approach where particular statements or expressions are tainted and monitored at run-time [37]. This analysis allows security practitioners to identify, for instance, potential attacker-controlled sources flowing into sensitive program regions. Yamaguchi et al. [53] define an over-approximate static approach by tainting program parts and propagating tainted values statically along the control and data flow. We extend their original formal definition to allow us to statically find vulnerabilities in patches per Definition 7.

Definition 7. *a) A taint-style analysis for vulnerable patch detection is a 4-tuple* $(V_{SOURCE}, V_{SINK}, V_{SAN}, V_{EDIT})$ *consisting of the nodes in the CCG of a program* P *denoting the taint source, sink and sanitizer nodes as depicted from* V_A *[53] as well as the nodes corresponding to code that is changed or newly created in a patch* $[P' \Rrightarrow P]$.

b) We say the patch $[P' \Rrightarrow P]$ *contains a bug if there exists a vulnerable data or control flow between any* $v_0 \in V_{SOURCE}$ *and* $v_1 \in V_{SINK}$ *with the constraint of not reaching any defined sanitizer but intersecting with at least one node from* V_{EDIT}.

4.1 Overview

To overcome the issues described in Section 3, our method is comprised of a new graph representation, called taint graphs which considers the context of a patch, a value-set analysis and an explainable graph neural network (GNN) that learns to infer detection rules on this particular representation in a taint-style fashion as described in Definition 7.

Graph representation. We define a new interprocedural patch graph representation. Beba and Karlsen [5] have shown that taint information significantly reduces false positives for rule-based static analyzers, hence, we similarly argue that this graph representation yields more valuable context to learning-based analyzers. In contrast to current discovery models, interprocedural graph representation is arguably more beneficial, since it enables us to propagate taint information within the entire program, which is impossible with a function, local slice or file-level graph.

Value-set analysis. As another improvement over recent discovery models, we calculate a value-set analysis to track variable domains in the graphs. This assists in reasoning about potential bounds and sanitizations. More specifically, whether or not the value of a user-controlled variable or buffer length is bounded beneficially affects the model's decision.

Causal GNN model. Finally, we use graph isomorphism network (GIN) layers to train an inductive model to infer detection rules applied to taint graphs. Our model is especially suited for processing long input graphs by its skip connections and attention mechanism. The attention weights from the latter can be interpreted as relevance scores per node to achieve a fine-granular localization of bugs.

4.2 Representation

To obtain a graph representation that is appropriate for vulnerability discovery in patches, we slice from *taint graphs* that are an extension to CCGs. We can calculate them in four steps:

(1) Insert call edges
(2) Insert interprocedural data flow
(3) Perform a value-set analysis
(4) Create security-relevant slices

(1) Insert call edges. A CCG is an intraprocedural disconnected graph G_{CCG} for a program P. Each function within P has its own CCG. We can connect each of them by adding call graph edges. More concretely, we connect call-sites with function definitions per Definition 4. We eventually end up with a single connected graph for P with $V = \bigcup_i^n V_A^i$ and $E = E_A \cup E_D \cup E_C \cup E_{CG}$. Although the CCG now encompasses an over-approximate global semantic of the program P with a single connected graph, it is still hard to track interprocedural data and control flow.

(2) Insert interprocedural data flow. The CG provides some intuition about the relation between functions during execution. Yet, to keep track of user-controlled variables, it is necessary to provide a more fine-grained view of the interprocedural data flow. Consider a function call $v_1 \rightarrow v_2$ with $(v_1, v_2) \in E_{CG}$. Since v_1 is a call statement, we can associate its accompanying argument nodes, while the same applies to the function parameters represented in the function signature of the callee v_2. We sort the argument nodes by their appearance in the function definition and connect them pairwise yielding E_{IDFG}. During the static analysis, it is hard to infer whether a variable passed by reference may be written to or only read from, thus, we model interprocedural data flow graph (IDFG) edges between pointers as a bidirectional relationship. This step leaves us with an IDFG graph formally defined in Definition 8.

Definition 8. *The interprocedural data-flow graph (IDFG) is defined as G_{IDFG} where $V_{IDFG} \subseteq \bigcup_{i=1}^{n} V_D^i$ with E_{IDFG} connecting parameters in the function call to their respective arguments in the function definition.*

(3) Perform a value-set analysis. Given G_D we can select any variable assignment $v_e \in V_D$ and find $(v_s, v_e) \in E_D$ where v_e reads from v_s. If v_s is a constant and v_e is a Boolean, Float or Integer operation, we can evaluate v_e. If v_s is not a constant, we can find $(v, v_s) \in E_D$ and repeat. This eventually boils down to constant propagation and folding. If we are able to evaluate v_e, we attach the evaluated value to the node. Otherwise, if the operation can not be evaluated because, for instance, one data flow dependent v_d of v_e relies on I/O input or external API calls, we annotate v_e with v_d.

Lastly as described by Wegman and Zadeck [50], we find all expressions within surrounding conditional blocks that may act as invariants. If within this conditional block, we run into a variable that appears in a conditional of the form `<var> <comparison> <expression>`, we annotate its bounds with its value if it could be evaluated in the previous step. As an example, in Figure 2, we can assert that `len` has a lower bound of 10 in the entire conditional block after line 4.

We can formalize this by attaching a lower-bound domain to every reference node $v \in V_D$ defined as a four-tuple semi lattice $(\mathbb{R}, \leq, \bot, \sqcap_l)$ and an upper-bound domain $(\mathbb{R}, \geq, \top, \sqcap_u)$. $\top$ and $\bot$ denote UNBOUNDED, that is, the variable has no upper or lower bound respectively, $\sqcap_u$ is the least upper bound defined as $\sqcap_u : (k_1, k_2) \rightarrow \min(k_1, k_2)$ and $\sqcap_l$, consequently, is the greatest lower bound defined as $\sqcap_l : (k_1, k_2) \rightarrow \max(k_1, k_2)$. Both are used as transfer functions at the control flow join points [3].

(4) Create security-relevant slices. At this point, we have a program P represented by an interprocedural CCG. As a first step towards the definition of taint graphs, we define taint paths. For this, we select taint sources $V_{\text{SOURCE}} \subset V$ providing user input and taint sinks $V_{\text{SINK}} \subset V$ denoting critical code regions and the subset $V_{\text{EDIT}} \subset V$ of all nodes that have been edited in a specific patch.

Definition 9. *A single* taint path *p of a patch $[P' \Rrightarrow P]$ is an oriented path with vertices $v_0, \ldots, v_b, \ldots v_e$ starting at $v_0 \in V_{SOURCE}$, passing through $v_b \in V_{EDIT}$ and ending in $v_e \in V_{SINK}$ where all the edges are in E_{IDFG}, E_{DFG} or E_{CFG}.*

To obtain taint paths we perform forward slicing from V_{EDIT} to V_{SINK} following any IDFG, DFG or CFG while neglecting the AST and CG edges. This leaves us with a set of paths that describe the changed spots within a patch potentially flowing into critical sinks. Likewise, we perform backward slices from V_{EDIT} to V_{SOURCE}. Combining both sets of slices leaves us with a set of paths describing all flows starting with user-defined inputs intersecting the patched locations and reaching the critical sinks. To provide a holistic view of a patch, we arrive at the taint graph, as defined in Definition 10, by combining all taint paths and gluing them together at their patch intersections V_{EDIT}. It is trivial to see, that this graph representation neatly fits into the definition of taint-style vulnerable patch detection from Definition 7.

Definition 10. *A taint graph (TG) of a patch $[P' \Rrightarrow P]$ is defined as G_{TG} joining its taint paths $\{p^1, p^2, \ldots, p^k\}$ at their common AST nodes, starting from V_{SOURCE} flowing through V_{EDIT} and reaching V_{SINK}.*

Originally, the term "taint graph" has been frequently used in malware analysis, where information collected by malicious applications is tracked to analyze how it flows through processes and files [55]. In our case, we are interested in how user-provided data flows through a patch and whether they may reach critical program sections. The definition of V_{SOURCE} and V_{SINK} is specific to the intended use and can be set appropriately. Furthermore, the number of paths in G_{TG} might become exponentially large, hence we suggest sub-sampling k paths at random.

4.3 Discovery

Let us consider the vulnerability in Figure 2 for CVE-2015-7497 which was introduced 12 years before it was publicly disclosed [2]. `plen` is a user-controlled variable that could trigger a buffer underflow in `name` when provided with an integer larger than `len`.

```
1 ...
2 value = *name;
3 value <<= 5;
4 if (len > 10) {
5     value += name[len - (plen + 1 + 1)];
6 ...
```

Figure 2: Buffer underflow in Libxml2.

The corresponding taint graph with $k = 4$ is depicted in Figure 3. For visualization purposes, we have omitted irrelevant node and edge labels and shortened the remaining node labels. We can see that four possible input sources flow into the array access to `name`. The common joint AST node, reached by each input source node happens to be a function call to `XMLDictLookUp()`. The critical node in this example is an array index calculation highlighted by PAVUDI.

Representation Learning. For a taint graph to be applicable for GNNs, we represent the textual code that is attached to every AST node using Word2Vec on each token and eventually take the average similar to recent works [11, 58]. We then train a causal graph isomorphism network (CGIN) [40] to infer bugs in patches using our

[2] https://github.com/GNOME/libxml2/commit/2fdbd3

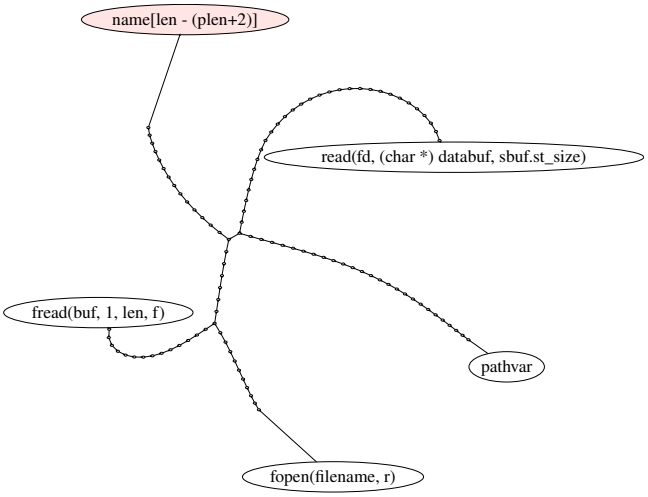

Figure 3: The G_{TG} with $k = 4$ for a Libxml2 buffer-underflow. Red denotes the relevant node according to PAVUDI.

novel graph representation. The GIN model performs state-of-the-art while relying on a simple update and aggregation mechanism similar to the GCN in Equation (1):

$$h^{(l)} = W^{(l-1)} \left((A + (1+\epsilon) \times I) \times \text{RELU}(h^{(l-1)}) \right) \tag{2}$$

Since, compared to classic code graphs, taint graphs are potentially longer and have a smaller average degree, we implement several design choices that help to pertain important information over large distances. We set ϵ in Equation (2) as a trainable parameter, which is particularly useful in conjunction with GINs to reduce smoothing out information from distant nodes. Furthermore, we use skip connections between the layers to help relevant information propagate across the topology.

After three graph encoding layers, we use two MLPs to calculate a relevance score for nodes and edges. For any node $v_i \in V$ we calculate their node attention as depicted in Equation (3) and for any pair of nodes $(v_i, v_j) \in E$ their edge attention as depicted in Equation (4) respectively. Furthermore, we halve the output space of the MLPs to perform a latent space disentanglement, where the first half will later be optimized to contain only the nodes causal for our task and the second half will be trained to only contain the trivial part of the graph which can be considered noise.

$$a_i^c, a_i^t = \sigma(\text{MLP}_{\text{NODE}}(h_i)) \tag{3}$$

$$b_{ij}^c, b_{ij}^t = \sigma(\text{MLP}_{\text{EDGE}}(h_i || h_j)) \tag{4}$$

A mean readout layer is applied last as a pooling strategy followed by a final MLP with softmax activation as the prediction head returning either VULNERABLE or CLEAN. Using the attention scores we can calculate attention masks $M_x, \hat{M}_x, M_a, \hat{M}_a$ respectively for the causal and trivial features, and causal and trivial edges. We apply these masks to the adjacency matrix and feature matrix of the taint graph yielding $\mathcal{G}^c$ and $\mathcal{G}^t$ for the causal and trivial subgraphs respectively. The causal taint graph can be used to explain the prediction and find the cause of a vulnerability.

To train the model in a supervised fashion we first apply a traditional NLL-loss $\mathcal{L}_{sup}$ to our ground truth and the latent representation of the causal graph h_{G_c} as depicted in Equation (5).

$$\mathcal{L}_{\text{sup}} = -\frac{1}{|\mathcal{D}|} \sum_{\mathcal{G} \in \mathcal{D}} \mathbf{y}_{\mathcal{G}}^{\top} \log\left(\sigma(\mathbf{h}_{\mathcal{G}^c})\right) \tag{5}$$

Then we take the representation of the trivial subgraph $h_{\mathcal{G}^t}$ and optimize the model to separate trivial and causal features by fitting the classifier with the trivial graph to be close to a uniform distribution using the Kullback-Leibler divergence (KL):

$$\mathcal{L}_{\text{unif}} = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{G} \in \mathcal{D}} \text{KL}\left(\mathbf{y}_{\text{unif}}, \sigma(\mathbf{h}_{\mathcal{G}^t})\right) \tag{6}$$

Ganz et al. [19] observe that common explanation methods put high relevance scores to features that stem from artifacts in the dataset. To reduce such bias taking effect in our future model interpretation, Sui et al. [40] suggest applying a backdoor adjustment to take care of any confounding variable. This can be achieved by conditioning the causal graph per sample with all possible trivial graphs $\mathcal{G}^t$ with $t \in \mathcal{T}$ obtained during training. This stabilizes training and helps reduce the influence of noise and spurious correlated features in the taint graph as defined in Equation (7).

$$\mathcal{L}_{\text{caus}} = -\frac{1}{|\mathcal{D}| \cdot |\mathcal{T}|} \sum_{\mathcal{G} \in \mathcal{D}} \sum_{t \in \mathcal{T}} \mathbf{y}_{\mathcal{G}}^{\top} \log\left(\sigma\left(\mathbf{h}_{\mathcal{G}^c} + \mathbf{h}_{\mathcal{G}^t}\right)\right) \tag{7}$$

After optimizing the model by minimizing $\mathcal{L}_{\text{sup}} + \mathcal{L}_{\text{unif}} + \mathcal{L}_{\text{caus}}$ we obtain a GNN that is able to process potentially long taint graphs. We can use the causal part of the graph G^c that was relevant to the model to classify the patch VULNERABLE and to localize the bug. Even more concretely using a_i^c from Equation (3) we can rank the most important causal nodes relevant for this classification, as the attention score can be directly interpreted as relevance score [36] e.g. by calculating the top-1 important node:

$$\max_{v_i \in V} a_i^c$$

4.3.1 Data Labeling. In order for a ML model to learn whether a patch potentially introduces new bugs, we would need to have a dataset with commits that originally add bugs to the code base, however, such a dataset is non-existent and not trivial to create, since, for example, a bug may need several commits until it manifests itself. Instead, current vulnerability datasets only contain commits that are known to fix bugs. Per commit, we can check out a software project and sample taint graphs with a pre-defined maximum length. We support the decision with a study from Calder et al. [8] stating that most C and C++ open-source projects have a maximum call-stack depth of 15. For each patch commit we mark the changed files and lines and move back in time to find commits prior to the patch touching the same lines in the same file. Particularly, for any commit including the patch, we extract its TG and label it VULNERABLE if it touches the same file and lines as the patch and label it CLEAN otherwise. This leaves us with three different types of TGs: Randomly selected clean graphs, vulnerable graphs that touch pre-fixed code locations, and clean graphs patching a bug. We will publish our extraction tool for subsequent research [3].

[3] https://github.com/SAP-samples/security-research-taintgraphs/tree/main/PAVUDI

In general, we can address the issues from Section 3, since the program-aware context-sensitive taint graphs can be regarded as a solution to the problem of *(1) context-sensitive changes*, since we are using only tainted paths we can consider the problem *(2) non-coherent changes* to be circumvented to some extent. Lastly, extracting the commits from different time steps and only if they are associated with a known bug addresses the issue *(3) evolution of software*. However, using this approach, we can't decide whether a patch introduces a bug, but whether patched code contains a bug. As long as we do not have a dataset with patches introducing bugs, but only patches that fix bugs, we argue, that this is a good compromise.

5 EVALUATION

In the following section, we first lay out our experimental setup and then provide answers to the following questions:

- RQ1 How do the individual components of PAVUDI contribute to the detection capability?
- RQ2 How do other strategies compare to PAVUDI?
- RQ3 How does the size of a commit affect the performance?
- RQ4 How does PAVUDI behave after training and deployment?

5.1 Experimental Setup

In this section, we present our experimental setup for the experimental evaluation.

5.1.1 Datasets. We use the state-of-the-art datasets from Zhou et al. [58] for comparison with related approaches. The datasets are derived from the open-source projects *FFmpeg*, a video decoding and encoding command line tool, and *QEMU*, a generic emulation software. Zhou et al. [58] curate a list of security-related keywords[4] associated with security patches that are used to crawl the Github repositories of both projects and find security-relevant commits. This has the advantage over other datasets [e.g. 17] that they have a large number of samples per project. They then proceed to extract code samples from the bug-fixing commits. We, on the other hand, use these fixing commits to extract clean taint graphs (at the time of the commit) and vulnerable taint graphs (prior to the commit) as outlined in Section 4.3.1. The FFmpeg project contains exactly 2558 vulnerable and 3037 clean or fixed taint graphs, while QEMU is slightly smaller with 1006 clean and 928 vulnerable taint graphs. We also assess PAVUDI's performance on five smaller projects, namely Libxml2, an XML parser, and Lrzip, a compression tool, since they have already been targeted by program analysis research [e.g. 7, 16, 20], cURL, a command line tool for HTTP requests, and OpenSSL, a cryptographic library, since these pose security-critical applications which have been exploited in the past and finally TinyProxy, as a relatively new untested security-relevant application.

5.1.2 Taint Graphs. Per commit, we sample at most $k = 1000$ taint paths with a maximum length of $l = 200$. For each commit, we can incrementally update the interprocedural CCG in a graph database using only the changes per patch. We choose V_{source} and V_{sink} similar to Pewny and Holz [34], by annotating *libc* functions that are related to bugs or provide user input. We refer the reader to the appendix for the complete list of tainted statements. Furthermore, we attach Boolean upper- and lower-bound information to each variable reference and assignment to provide the model with information on whether the value of the expression is bounded.

5.1.3 Baseline Models. There are several state-of-the-art methods for learning-based static vulnerability discovery. We compare against three intraprocedural graph-learning-based models since they also rely on graph representations. These models classify bugs only on the function level.

Devign uses as its initial input graph the CPG enhanced with the natural sequence graph connecting leaf nodes in their order of evaluation [58]. The model consists of a six-step GGNN with a one-dimensional CNN as pooling.

ReVeal is similar to Devign and uses an eight-step GGNN to embed the graph structure in latent space [11]. However, they use a simple max pooling. Their original input graph is the CPG and they use a triplet loss for training.

BGNN4VD uses the bidirectional CCG instead of the CPG [9]. They use an eight-step GGNN just as ReVeal followed by a 1D convolutional pooling and a final linear layer.

Due to the fact that all three only consider local functions, their static analysis approach is not context-sensitive, however flow-sensitive since they use data and control flow edges [39]. Thus, as another set of baselines, we select two models that are able to process bugs in a limited interprocedural context.

DeepWukong is a graph-based learning model [14]. Compared to the other GNN-based approaches it uses pre-processed slices around potentially critical code locations, for instance, array indexing arithmetics, pointer usages or library calls.

SysEVR is similar to DeepWukong, as it first finds locations potentially containing bugs, for instance, pointer usages, arithmetic expressions, function calls, and array indexing [29]. However, instead of using a graph representation, they stick to the token representation of the code extracted from the slicing operation on the CPG to feed it into a bidirectional gated recurrent unit (BGRU).

DeepWukong and SySEVR extract syntactic vulnerability candidates in the source code used for positioning the slices. Their slicing operation includes function calls within the function under analysis. Both approaches are context-sensitive and flow-sensitive since they either use flow graphs or a flow-sensitive slicing approach. Lastly, we compare PAVUDI's performance against two more popular approaches.

VUDDY [25] is a non-learning-based static analyzer that detects bugs by comparing their function signatures against known CVEs and NVDs.

VulDeePecker similarly to SySEVR uses a token-based representation of intraprocedural forward and backward slices over the CPG [30]. It uses a BiLSTM for classification.

VUDDY is a non-learning-based analyzer that is not optimized for the dataset and thus a suitable baseline representative for other non-learning based SAST tools. VulDeePecker, a context-unaware analyzer, is similar to SySEVR but does not incorporate any a priori

[4] https://sites.google.com/view/devign

information about the slicing locations. Furthermore, VUDDY is neither context nor flow-sensitive.

5.1.4 Strategies. All baselines process code at different granularity, but to the best of our knowledge, none of them has been previously applied to patches. While some methods classify slices and others' entire functions, it remains unclear how one would apply them to patches. Thus, we aggregate their decisions to a single score per patch using five different aggregation strategies as if we would integrate them into a software quality assurance process:

Max is a strategy where simply the maximum value of all prediction scores from slices or functions is taken.

Mean strategy averages over every prediction score from slices or functions.

Probability is a strategy where the probability of the patch being vulnerable depending on its k components, is similar to calculating a system's failure probability.

$$P = 1 - \prod_{i=0}^{k} (1 - f(p_i)) \tag{8}$$

As denoted in Equation (8), for each vulnerable component, we multiply the probability of the complement event yielding the probability that no component is vulnerable. Then we take the complementary event again and obtain the probability that at least one component is flawed.

Isotonic Probability is similar to the probability strategy. Niculescu-Mizil and Caruana [31] state that prediction scores from ML models can not be mapped to probabilities out-of-the-box. We use an isotonic regressor to calibrate the predictions before calculating the probability as in Equation (8).

Commit merges all changed code components within a patch together and feeds them into the model if applicable, instead of returning predictions per function.

The different strategies have different effects on the FPs and TPs. If the vulnerable score for the only function in the patch gets smoothed out with the Mean-Strategy, we have fewer TPs and FPs. On the other hand, the Max-Strategy may be too sensitive to functions being slightly above the threshold resulting in higher FPs and TPs resulting in a lower precision but higher recall. Note that VUDDY only returns *CLEAN* or *VULNERABLE* without any confidence score. Hence, we can only apply strategy Commit and Max. The slice-based approaches can not have merged components as input hence we can not apply the Commit-Strategy to them.

5.1.5 Performance Metrics. To assess the performance of the different models we use several performance measurements that are recommended for comparing ML models in security [4]. For the comparison against other models and in the ablation study we use the F1-Score, that is, the harmonic mean between precision and recall. Furthermore, we use the area under receiver operating characteristic curve (AUROC), particularly as a second measurement for the evaluation against the baselines. For the concept drift experiment, we choose the balanced accuracy, calculated as the arithmetic mean between the sensitivity and specificity. We repeat every experiment ten times and report the best score.

Table 1: Ablation study: F1-Scores measured for different settings.

Dataset	GIN	GGNN	GCN	CGIN
FFmpeg+CutOff	0.42 ± 0.14	0.48 ± 0.08	0.50 ± 0.10	0.56 ± 0.05
QEMU+CutOff	0.51 ± 0.11	0.47 ± 0.08	0.50 ± 0.03	0.61 ± 0.22
FFmpeg+TG	0.89 ± 0.02	0.89 ± 0.01	0.85 ± 0.02	0.90 ± 0.02
QEMU+TG	0.84 ± 0.01	0.79 ± 0.03	0.79 ± 0.02	0.84 ± 0.01
FFmpeg+TG+Bounds	0.90 ± 0.03	0.89 ± 0.01	0.89 ± 0.02	**0.91 ± 0.02**
QEMU+TG+Bounds	0.84 ± 0.03	0.80 ± 0.03	0.80 ± 0.02	**0.85 ± 0.21**

5.1.6 Implementation Details. We implement PAVUDI on top of Pytorch Geometric and Memgraph[5] for storing and transforming the taint graphs. Furthermore, we use an AWS EC2 g4dn instance for extracting the taint graphs and for training. We use PyDriller and the GitHub API for extracting patch information, such as for instance, the commit date, the number of changed functions and methods. We train every model including PAVUDI on the same dataset with an identical 70/30 random split, except for the non-learning-based static analyzer VUDDY. We use the hyperparameters from the respective original publications or reference implementations for the baselines. For PAVUDI we use an ADAM optimizer with a learning rate of 0.0001 and for the Word2Vec node embedding we use a vector of size 100 and a context window of size 3 [11].

5.2 Evaluation

We proceed to present our experimental results to provide answers to our research questions.

How do the individual components of PAVUDI contribute to the detection capability? We present our ablation study for our CGIN model on taint graphs on the FFmpeg and QEMU datasets. We try three other popular GNN architectures, namely GIN [52], GGNN [28] and GCN [26]. Also, we evaluate the models without using the bounds information from the value-set analysis. Finally, we slice off a subgraph from the taint graphs, such that each graph only captures the immediate data flow around the patch neglecting taint information. In Table 1, we see that the cut-off taint graphs yield worse performance, hence we argue that providing interprocedural and taint information is crucial for classifying vulnerable patches. GCN is the worst architecture, while GGNN is very close to GIN. However, CGIN provides the best F1-Scores.

Table 2: F1-Score for dataset cross-evaluation.

Testset	Trainset		
	FFmpeg	QEMU	FFmpeg+QEMU
FFmpeg	**91.1** ± 1.7%	34.0 ± 1.4%	N/A
QEMU	41.3 ± 1.1%	**82.9** ± 2.7%	N/A
Libxml2	39.6 ± 1.8%	**58.0** ± 1.2%	57.0 ± 1.6%
cURL	**48.5** ± 0.3%	22.2 ± 2.2%	14.0 ± 1.2%
OpenSSL	60.9 ± 1.43%	**83.4** ± 1.6%	54.0 ± 2.4%

Using the backdoor adjustment from Sui et al. [40] our results align with their observation, that we can even reduce the out-of-distribution (OOD) problem. This can be seen in Table 2 where PAVUDI achieves noticeable detection performance measured by

[5] https://memgraph.com/

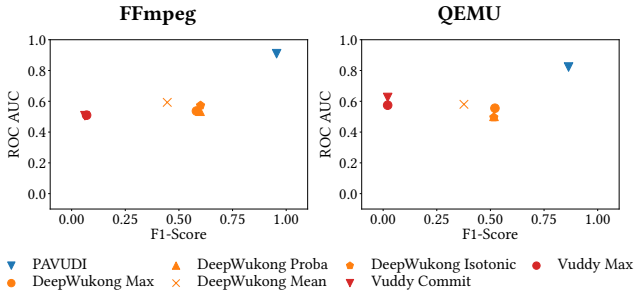

Figure 4: Performance comparison against DeepWukong and Vuddy.

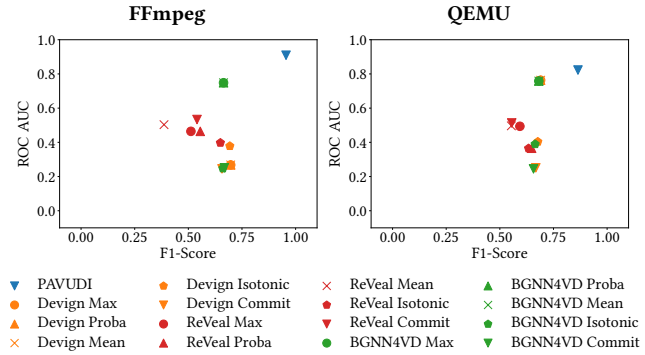

Figure 5: Performance comparison against Devign, ReVeal and BGNN4VD.

the F1-Scores on completely different open-source software projects that it was not trained on. It achieves an F1-Score of 58% for Libxml2 and even 83.4% for OpenSSL, although, there is an overlap between QEMU's and OpenSSL cryptographic feature implementations.

> Context information contained in taint graphs and GIN layers significantly contributes to the detection performance of PAVUDI.

How do other strategies compare to PAVUDI? We compare our approach against the seven baselines with the five aggregation strategies. In Figure 4 we can see that PAVUDI has a much higher AUROC and F1-Score compared to Vuddy and DeepWukong. Since VUDDY is a deterministic method and not fine-tuned to our dataset, the bad performance is expected and is representative of other rule-based SAST tools. DeepWukong, however, is a sliced-based GNN approach and only achieves an F1-Score of 65% using the Isotonic-Strategy. In Figure 8 it is surprising that the simple token-based approach VulDeePecker with the Probability-Strategy achieves an AUROC of 79% on QEMU and Devign beating SySEVR. Furthermore, we observe an overall beneficial score using the isotonic projection for all methods. Figure 5 shows the result against other graph-based methods. Especially BGNN4VD outperforms the former token-based and slice-based methods with an AUROC of 80% and an F1-Score of 75%. In all of our experiments, using the Mean-Strategy yields the worst scores. The Max and Commit Strategies are similarly underperforming. Both increase the false positive rate too much resulting in disadvantageous F1- and AUROC-scores. That means the naive strategy, to merge all changed functions and classify this, is detrimental to the detection performance. However, calculating the failure probability using the Probability-Strategy is the best aggregation approach. The inferiority of SySEVR and DeepWukong may stem from the fact that they define slices around syntactic vulnerability candidates which may end up with too many candidates and foster false positive alerts. 70% of all statements in FFmpeg correspond to syntactic vulnerable candidates using SySEVR potentially posing a low signal-to-noise ratio.

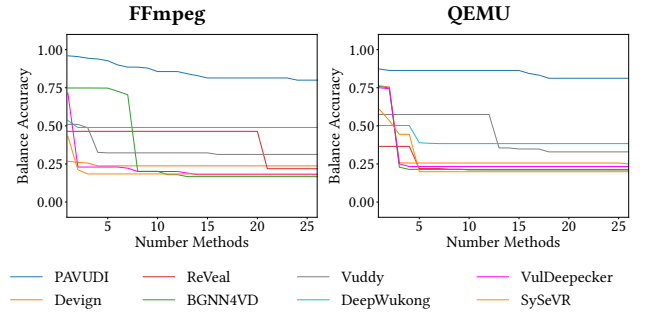

Figure 6: Performance decrease with increasing number of changed methods.

> The probabilistic aggregation is the best strategy for previous models applied to patches, still, PAVUDI provides an up to three times stronger detection performance.

How does the size of a commit affect the performance? We assume that the difficulty of detecting bugs is positively correlated with number of methods touched in a commit. Relying on the Mean-Strategy, for instance, would smooth out the result of a bug prediction with too many changed methods. Resulting in a larger amount of false negatives. Since the Probability-Strategy yields the best result in our evaluation, we proceed to measure the performance loss with commits that have an increasing number of changed methods. In Figure 6 we indeed observe a performance drop. With five changed methods the model performance already deteriorates significantly. VUDDY and BGNN4VD can pertain to their original performance only when analyzing less than 13 and 8 methods respectively. PAVUDI, however, is only slightly affected by the number of methods in a commit.

> The detection performance of most baselines deteriorates after 8 changed methods within a patch. PAVUDI's performance slightly drops only after 15 methods.

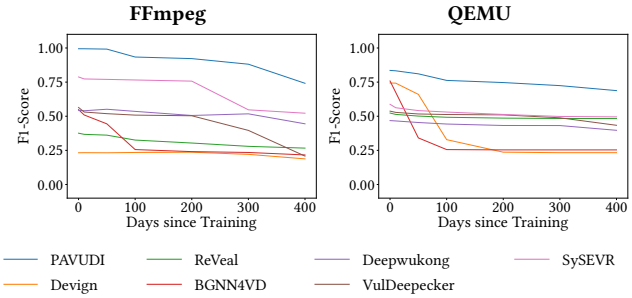

Figure 7: Performance decrease measured over time since training.

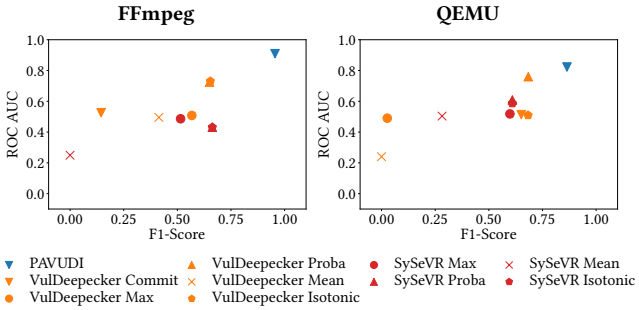

Figure 8: Performance comparison against VulDeePecker and SySEVR.

How does PAVUDI behave after training and deployment? Concept drift describes the performance loss after the deployment of a model. We assume, that the features that a model is applied on might change relative to the features the model was trained on originally. We conduct another experiment to explain whether a model needs to be fine-tuned after initial training and how large its generalization effect is. We train all models, excluding VUDDY, on our dataset. The dataset is sorted by contribution date and split in half. The first half is for training and the second is for validation. In Figure 7, we see that PAVUDI has a significant drop after 300 days. DeepWukong deteriorates after 200 days on FFmpeg. The other models struggle to learn anything initially on the time-sorted datasets. On QEMU only Devign and BGNN4VD have a competitive initial F1-Score of 75%. However, their detection capabilities shrink already after respectively 25 and 60 days.

> All baseline models lose their initial performance after at most half a year from training. Only after one year, PAVUDI significantly loses its detection efficiency.

6 PRACTICAL APPLICATION

We demonstrate that PAVUDI has a notable performance benefit over other learning-based vulnerability discovery methods when applied to patches. Due to this, we propose to integrate PAVUDI in practical scenarios.

Integration. PAVUDI is specifically designed to detect patches that either introduce or touch bugs or security vulnerabilities. As already detailed in Section 4.3.1, we can detect bugs in patches by checking out a project at a specific commit and extracting a taint graph through the changed code lines with respect to the preceding commit. This effectively enables us to integrate PAVUDI into a secure software development lifecycle. For instance, PAVUDI can be triggered on every new commit within a continuous integration or deployment pipeline and analyze the changes. We report a finding if PAVUDI's confidence score for a code change exceeds a threshold. However, due to the inherent problem of concept drift in the vulnerable patch detection task which we have shown in our experiments, it is evident that PAVUDI requires a continuous learning process when deployed in software development. Therefore, PAVUDI has to be retrained or fine-tuned regularly.

Interpretation. Each reported finding should be reviewed manually. Therefore, it is essential that the results help security practitioners to find the root cause of the bug quickly. For this reason, we have added an explanation mechanism to PAVUDI that can highlight relevant nodes in the taint graph. Recall the vulnerable patch from Figure 2 and its taint graph in Figure 3 with its most important node highlighted in red according to PAVUDI. Just like Sui et al. [40], we arrive at such an explanation by considering the causal graph G^c during inference. Since the attention mask can be interpreted as relevance scores, we can rank the nodes by their attention and highlight the node with the largest attention value. This directly hints us to the node responsible for an array index calculation that causes the bug in Figure 3. However, taint graphs can become very large and cluttered, thus we suggest extracting line-level information from the highlighted relevant nodes. By simply storing the line-level information on every node, we can extract the relevant code lines from the highlighted nodes according to the explanation. This allows for precise localization of bugs and vulnerabilities and may even be integrated into integrated development environments (IDEs).

6.1 Case-Studies

To demonstrate the practicability of our tool, we apply it in a realistic scenario on two tools with their most recent 100 commits at the time of writing, namely *Tinyproxy* and *Lrzip* to find unknown bugs. We reported every finding to the respective maintainer. In particular, we proceed as outlined in the prior section: We extract the taint graphs, run PAVUDI's inference and extract line-level explanation scores.

Table 3: Detected bugs by PAVUDI.

Project	Bug Description	Found in Commit	Fixed?
Tinyproxy	Buffer-Overflow	453235	Fixed (470cc0)
TinyProxy	Undefined Behavior	64badd	Fixed (6ffd9a)
TinyProxy	Missing Format Limits	252959	Fixed (3764b8)
TinyProxy	Undefined Behavior	252959	Reported
Lrzip	Use-Of-Uninitialized Memory	09ceb8	Reported

Overall we found five bugs as depicted in Table 3. In TinyProxy we identified a buffer overread, three bugs related to undefined behavior, and one bug related to missing width limits in `sscanf`

fields. In Lrzip we found a use-of-uninitialized value. We present a bug found in TinyProxy, an HTTP/HTTPS proxy written in C, as a case study in the following Section. In Figure 9, we can see the change of a config parser function. The analyzed patch should initially enhance the parsing speed, however, PAVUDI detects a bug at line 11. The pointer `q` might exceed its limit resulting in a buffer over-read. TinyProxy expects its configuration input space-separated key-value-pairs to be of the form `key value`. If a space is missing, the program crashes.

```
1  ...
2  - while (fgets (buffer, sizeof (buffer), f)) {
3  -         if (check_match (conf, buffer, lineno)) {
4  -             printf ("Syntax error on line %ld\n", lineno);
5  + for (;fgets (buffer, sizeof (buffer), f);++lineno) {
6  +         if(buffer[0] == '#') continue;
7  +         p = buffer;
8  +         while(isspace(*p))p++;
9  +         if(!*p) continue;
10 +         q = p;
11 +         while(!isspace(*q))q++;
12           ...
```

Figure 9: The buffer overflow in TinyProxy's config parser.

6.2 Limitations

The discovery of security vulnerabilities in software is a hard problem that is undecidable in the general case. As a result, our approach naturally comes with limitations and blind spots that we discuss in the following.

Non-taint vulnerabilities. PAVUDI relies on data and control flow between user-controlled sources and critical sinks. This consequently means it can only detect vulnerabilities that manifest in the interprocedural data or control flow. We cannot possibly find bugs that do not share this characteristic, for instance, race conditions or deadlocks.

Definition of sinks. The performance of our approach depends on the definition of appropriate sinks and sources. This may be tricky and subject to each individual project. We follow the approach by Pewny and Holz [34] as described in Section 5.1.2. While this selection of *libc* functions can miss certain vulnerabilities, such as off-by-one errors, including all pointer arithmetics and function calls, as Li et al. [29] propose, leads to computationally infeasible large graphs for our method.

Model updates. Although PAVUDI seems to be more stable than other methods, we see a performance decline after model deployment. This indicates that it is necessary to regularly update or retrain PAVUDI with new data. Therefore, we propose to combine PAVUDI with other vulnerability discovery methods, such as rule-based and dynamic analysis methods, so that each balances the blind spots of the others until updates to models and rules are available.

7 RELATED WORK

In this section, we present related literature from research areas tangent to this work.

Learning-based vulnerability discovery. Several past works already target the problem of automatically discovering vulnerabilities and bugs. For instance, Russell et al. [35] and Li et al. [30] use a token-based approach. The combination of GNNs and code graphs has already been proven to be suited for the discovery of bugs and security vulnerabilities in software [10, 13, 44, 59]. Zhou et al. [59] introduce the first gated graph neural network on code property graphs to identify bugs in vulnerabilities collected from real-world commits. Their approach outperforms popular open-source and commercial static analyzers as well as token-based ML models. Cao et al. [10] combine data and control-flow graphs with the abstract syntax tree to the code composite graph.

Interprocedural Graphs. Li et al. [29] use interprocedural slices for the vulnerability discovery task. Zheng et al. [57] show that graphs extracted from intraprocedural function slices make it impossible for the models to learn interprocedural bugs spanning multiple functions. While these approaches try to incorporate interprocedural information, it is insufficient since either the function call depth is limited and rather small, or neither the input source nor critical sinks are considered. Using a whole-program interprocedural graph representation would solve this problem, however, this is a non-trivial task, since programs can become rather large [41]. Our approach overcomes the issue by only selecting relevant paths. Cheng et al. [15] extract multiple interprocedural paths starting from a function under analysis until its return. They neglect sources and sinks and abstain from a whole-program perspective.

Learning on patches. Since this work focuses on patches, it is noticeable that there is a research interest around the classification of patches [46, 48] that have been introduced silently or are security relevant. In this particular field, GNNs could have been applied successfully [45]. Wang et al. [43] also classify security-relevant patches to improve the dataset preparation. Another research branch tries to detect anomalies in patches using meta-information of the specific versioning control system [18, 21, 22].

Explainable AI. Deep-Learning models tend to be black boxes, hence, there exist a large variety of explanation methods to enable us to interpret a model's decision. Guo et al. [24] introduce a black-box method specifically for security-relevant models. Selvaraju et al. [38], for instance, introduce a white-box method for image classification. Sanchez-Lengeling et al. [36] port many methods to the graph domain. Ganz et al. [19] and Warnecke et al. [49] show that explanation methods used to locate bugs tend to reveal bias and artifacts from the training procedure of a model. We address the explainability of PAVUDI in this paper using soft attention as presented by Sanchez-Lengeling et al. [36] as it has already been done in other works [17].

8 CONCLUSION

In this work, we adapt several state-of-the-art learning-based vulnerability discovery models to vulnerable patch detection and show that they have a poor performance and their detection capabilities even degrade over time after deployment. As a consequence, we present our novel patch-based vulnerability detection model, PAVUDI, which leverages interprocedural code graphs and taint-style static program analysis.

ACKNOWLEDGMENTS

This work has been funded by the Federal Ministry of Education and Research (BMBF, Germany) in the project IVAN (FKZ: 16KIS1165K).

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *CoRR* abs/1711.00740 (2017). arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[3] Roberto Amadini, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2020. Abstract Interpretation, Symbolic Execution and Constraints. In *Recent Developments in the Design and Implementation of Programming Languages (OpenAccess Series in Informatics (OASIcs), Vol. 86)*, Frank S. de Boer and Jacopo Mauro (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:19. https://doi.org/10.4230/OASIcs.Gabbrielli.7

[4] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2020. Dos and Don'ts of Machine Learning in Computer Security. *CoRR* abs/2010.09470 (2020). arXiv:2010.09470 https://arxiv.org/abs/2010.09470

[5] Sindre Beba and Magnus Melseth Karlsen. 2019. Implementation Analysis of Open-Source Static Analysis Tools for Detecting Security Vulnerabilities.

[6] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *CoRR* abs/1806.07336 (2018). arXiv:1806.07336 http://arxiv.org/abs/1806.07336

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[8] Brad Calder, Dirk Grunwald, and Benjamin Zorn. 1994. Quantifying Behavioral Differences Between C and C++ Programs. *Journal of Programming Languages* 2 (02 1994).

[9] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Inf. Softw. Technol.* 136 (2021), 106576.

[10] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Information and Software Technology* 136 (2021), 106576. https://doi.org/10.1016/j.infsof.2021.106576

[11] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[12] Yizheng Chen, Zhoujie Ding, Xinyun Chen, and David A. Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. *ArXiv* abs/2304.00409 (2023).

[13] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2020. DeepWukong: Statically Detecting Software Vulnerabilities using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol. 1, 1, Article* 1 (2020), 32. https://doi.org/10.1145/3436877

[14] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 38 (apr 2021), 33 pages. https://doi.org/10.1145/3436877

[15] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 519–531. https://doi.org/10.1145/3533767.3534371

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[17] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE.

[18] Tom Ganz, Inaam Ashraf, Martin Härterich, and Konrad Rieck. 2023. Detecting Backdoors in Collaboration Graphs of Software Repositories. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy* (Charlotte, NC, USA) *(CODASPY '23)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/3577923.3583657

[19] Tom Ganz, Martin Härterich, Alexander Warnecke, and Konrad Rieck. 2021. Explaining Graph Neural Networks for Vulnerability Discovery. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security* (Virtual Event, Republic of Korea) *(AISec '21)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/3474369.3486866

[20] Tom Ganz, Philipp Rall, Martin Härterich, and Konrad Rieck. 2023. Hunting for Truth: Analyzing Explanation Methods in Learning-based Vulnerability Discovery. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 524–541. https://doi.org/10.1109/EuroSP57164.2023.00038

[21] Izhak Golan and Ran El-Yaniv. 2018. Deep anomaly detection using geometric transformations. *arXiv preprint arXiv:1805.10917* (2018).

[22] Danielle Gonzalez, T. Zimmermann, Patrice Godefroid, and Maxine Schaefer. 2021. Anomalicious: Automated Detection of Anomalous and Potentially Malicious Commits on GitHub. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2021), 258–267.

[23] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New Orleans, Louisiana, USA) *(CODASPY '16)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2857705.2857720

[24] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. LEMNA: Explaining Deep Learning Based Security Applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 364–379. https://doi.org/10.1145/3243734.3243792

[25] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[26] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

[27] Dexter C. Kozen. 1977. *Rice's Theorem*. Springer Berlin Heidelberg, Berlin, Heidelberg, 245–248. https://doi.org/10.1007/978-3-642-85706-5_42

[28] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2017. Gated Graph Sequence Neural Networks. arXiv:1511.05493 [cs.LG]

[29] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. 2018. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *CoRR* abs/1807.06756 (2018). arXiv:1807.06756 http://arxiv.org/abs/1807.06756

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *CoRR* abs/1801.01681 (2018). arXiv:1801.01681 http://arxiv.org/abs/1801.01681

[31] Alexandru Niculescu-Mizil and Rich Caruana. 2005. Predicting good probabilities with supervised learning. *ICML 2005 - Proceedings of the 22nd International Conference on Machine Learning*, 625–632. https://doi.org/10.1145/1102351.1102430

[32] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[33] Cong Pan and Michael Pradel. 2021. Continuous Test Suite Failure Prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 553–565. https://doi.org/10.1145/3460319.3464840

[34] Jannik Pewny and Thorsten Holz. 2016. EvilCoder: Automated Bug Insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 214–225. https://doi.org/10.1145/2991079.2991103

[35] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *CoRR* abs/1807.04320 (2018). arXiv:1807.04320 http://arxiv.org/abs/1807.04320

[36] Benjamin Sanchez-Lengeling, Jennifer Wei, Brian Lee, Emily Reif, Peter Wang, Wesley Qian, Kevin McCloskey, Lucy Colwell, and Alexander Wiltschko. 2020. Evaluating Attribution for Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 5898–5910. https://proceedings.neurips.cc/paper/2020/file/417fbbf2e9d5a28a855a11894b2e795a-Paper.pdf

[37] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. 317–331. https://doi.org/10.1109/SP.2010.26

[38] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.

[39] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-Sensitive Data-Flow Analysis Using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (jan 2019), 29 pages. https://doi.org/10.1145/3290361

[40] Yongduo Sui, Xiang Wang, Jiancan Wu, Min Lin, Xiangnan He, and Tat-Seng Chua. 2022. Causal Attention for Interpretable and Generalizable Graph Classification. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery*

and Data Mining (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 1696–1705. https://doi.org/10.1145/3534678.3539366

[41] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental Whole-Program Analysis in Datalog with Lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3453483.3454026

[42] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. *International Conference on Learning Representations* (2018). https://openreview.net/forum?id=rJXMpikCZ

[43] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. 2021. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[44] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2021. Combining Graph-Based Learning with Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[45] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*. 604–621. https://doi.org/10.1109/SP46215.2023.00035

[46] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. PatchDB: A Large-Scale Security Patch Dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 149–160. https://doi.org/10.1109/DSN48987.2021.00030

[47] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. PatchRNN: A Deep Learning-Based System for Security Patch Identification. In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 595–600. https://doi.org/10.1109/MILCOM52596.2021.9652940

[48] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. PatchRNN: A Deep Learning-Based System for Security Patch Identification. In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 595–600. https://doi.org/10.1109/MILCOM52596.2021.9652940

[49] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. 2020. Evaluating Explanation Methods for Deep Learning in Security. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Genoa, Italy, 158–174. https://doi.org/10.1109/EuroSP48549.2020.00018

[50] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (apr 1991), 181–210. https://doi.org/10.1145/103135.103136

[51] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[52] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ryGs6iA5Km

[53] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[54] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[55] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. *Proceedings of the ACM Conference on Computer and Communications Security*, 116–127. https://doi.org/10.1145/1315245.1315261

[56] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 26–36. https://doi.org/10.1145/2025113.2025121

[57] Weining Zheng, Yuan Jiang, and Xiaohong Su. 2021. VulSPG: Vulnerability detection based on slice property graph representation learning. *CoRR* abs/2109.02527 (2021). arXiv:2109.02527 https://arxiv.org/abs/2109.02527

[58] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *CoRR* abs/1909.03496 (2019). arXiv:1909.03496 http://arxiv.org/abs/1909.03496

[59] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *CoRR* abs/1909.03496 (2019). arXiv:1909.03496 http://arxiv.org/abs/1909.03496

APPENDIX

Table 4: Functions for V_{source}.

Function	Description
getchar/getc/getch	Reads a char from stdin
fgets	Reads a line from a stream
read	Reads from a file descriptor
fopen	Opens a file
scanf	Reads formatted input from stdin
gets	Reads input from stdin
fscanf	Reads formatted input from a stream
getenv/secure_getenv	Reads from an environment variable
fread	Reads input from a stream
poll/ppoll	Wait for event on file descriptor
recvfrom/recv/recvmsg	Receives message from socket

Table 5: Functions for V_{sink}.

Function	Description
malloc/calloc/realloc	Allocate heap memory.
memcpy	Copies memory content.
strcpy	Copies string content.
printf/snprintf/sprintf	Provides formatted output.
memset	Initializes memory.
strcat	Concatenates strings.
free	Deallocates memory.