Snopy: Bridging Sample Denoising with Causal Graph Learning for Effective Vulnerability Detection

Sicong Cao
Yangzhou University
Yangzhou, China
DX120210088@yzu.edu.cn

Xiaobing Sun*
Yangzhou University
Yangzhou, China
xbsun@yzu.edu.cn

Xiaoxue Wu
Yangzhou University
Yangzhou, China
xiaoxuewu@yzu.edu.cn

David Lo
Singapore Management University
Singapore, Singapore
davidlo@smu.edu.sg

Lili Bo†
Yangzhou University
Yangzhou, China
lilibo@yzu.edu.cn

Bin Li
Yangzhou University
Yangzhou, China
lb@yzu.edu.cn

Xiaolei Liu
China Academy of Engineering Physics
Mianyang, China
luxaole@gmail.com

Xingwei Lin
Zhejiang University
Hangzhou, China
xwlin.roy@zju.edu.cn

Wei Liu
Yangzhou University
Yangzhou, China
weiliu@yzu.edu.cn

ABSTRACT

Deep Learning (DL) has emerged as a promising means for vulnerability detection due to its ability to automatically derive features from vulnerable code. Unfortunately, current solutions struggle to focus on vulnerability-related parts of vulnerable functions, and tend to exploit spurious correlations for prediction, thus undermining their effectiveness in practice. In this paper, we propose Snopy, a novel DL-based approach, which bridges sample denoising with causal graph learning to capture real vulnerability patterns from vulnerable samples with numerous noise for effective detection. Specifically, Snopy adopts a change-based sample denoising approach to automatically weed out vulnerability-irrelevant code elements in the vulnerable functions without sacrificing the label accuracy. Then, Snopy constructs a novel Causality-Aware Graph Attention Network (CA-GAT) with Feature Caching Scheme (FCS) to learn causal vulnerability features while maintaining efficiency. Experiments on the three public benchmark datasets show that Snopy outperforms the state-of-the-art baselines by an average of 27.22%, 85.89%, and 75.50% in terms of F1-score, respectively.

*Xiaobing Sun is the corresponding author.
†Yunnan Key Laboratory of Software Engineering, Yunnan, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695057

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Computing methodologies** → *Artificial intelligence.*

KEYWORDS

Program Analysis, Causal Learning, Graph Attention Network

ACM Reference Format:
Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, Xiaolei Liu, Xingwei Lin, and Wei Liu. 2024. Snopy: Bridging Sample Denoising with Causal Graph Learning for Effective Vulnerability Detection. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695057

1 INTRODUCTION

Software vulnerabilities are security-related flaws introduced during the design and implementation of the software. Such weaknesses could be exploited by a threat actor for a variety of malicious ends [31]. For example, `XZ-Utils`[1], a data-compression library widely integrated into Linux ecosystem, was recently disclosed to contain a backdoor (CVE-2024-3094[2]), enabling a remote attacker to execute arbitrary code. Thus, despite significant efforts to enhance software reliability in the past few decades, vulnerability detection remains a classic yet challenging problem.

Due to the high flexibility for analyzing software without running it, static code analyzers [2, 6, 9, 23] are widely used to hunt security vulnerabilities hidden in programs. However, state-of-the-art tools have achieved limited success in realistic scenarios as they heavily rely on hand-crafted vulnerability specifications and rules, which are *time-consuming* and *error-prone* [40]. Benefiting from the construction of large-scale vulnerability datasets [10, 20, 46] and tremendous progresses of Deep Learning (DL) in code understanding, recent years have witnessed an increasing in the popularity of

[1]https://tukaani.org/xz/
[2]https://nvd.nist.gov/vuln/detail/CVE-2024-3094

learning-based vulnerability detection approaches [11, 38, 39, 71]. Compared to conventional Program Analysis (PA)-based solutions, DL-based approaches can automatically learn implicit vulnerability patterns from source code without human intervention.

While promising, current neural vulnerability detectors suffer from severe performance degradation when applied to detect vulnerabilities in real-world settings, posing a barrier to adoption [8]. Such poor generalization ability stems from two major **challenges**:

Challenge 1: Noisy program semantics. Since acquiring a reliable vulnerability label is a non-trivial task, previous neural vulnerability detectors [3, 36, 59, 71] primarily extract features from the entire functions, instead of vulnerability-related code regions. As a result, DL models will be heavily influenced by the program semantics of vulnerability-irrelevant code elements (i.e., *noise*) in the vulnerable samples. To alleviate this problem, a straightforward countermeasure is program slicing [60], which prunes irrelevant statements starting from a Point of Interest (PoI), as recent works [4, 38, 39] do. Unfortunately, such pre-processing still requires specialized security expertise (e.g., slicing criteria), and is limited to specific types of vulnerabilities [63]. Moreover, since a considerable portion (e.g., accounting for 24.19% in the Big-Vul [20] dataset) of vulnerabilities are fixed by adding compulsory security checks without any deletion, existing deletion-based annotation rules (i.e., a slice is considered to be vulnerable as long as it covers at least one vulnerable statement removed in the security patch [11]) may not work and even introduce additional data quality issues [15].

Challenge 2: Learning spurious correlations. Recent works [37, 66] pointed out that, both traditional Deep Neural Network (DNN)-based and emerging Large Language Model (LLM)-based vulnerability detectors can be easily fooled into flipping their decisions by simple perturbations such as identifier replacement. This is because DL models tend to pick up dataset nuances (e.g., specific coding styles) for prediction [56], as opposed to causal vulnerability features (e.g., potential root causes and manifestation points) [43]. To cut off the correlations between spurious features and model outputs, state-of-the-art approaches either retrain detection models on obfuscated program variants [7] or ignore the spurious features during the inference phase [50]. For example, Rahman et al. [50] assumed variable and API names can be abused as spurious features, and prevented them from participating in decision-making. Nevertheless, current solutions focus on enhancing the robustness of detection models against target samples with spurious features, instead of capturing real vulnerability patterns, resulting in marginal performance improvements.

Our Work. Based on the above discussion, an important question arises as *"how to capture real vulnerability patterns from vulnerable samples with numerous noise for effective detection?"* In response, we propose Snopy, a novel DL-based approach, which combines **S**ample de**NO**ising with causal gra**P**h learning for effective vulnerabilit**Y** detection. The key insights underlying our approach include (❶) patch modifications aim to sanitize or interrupt vulnerable program behaviors, and naturally can be utilized as *explicit expert knowledge* to localize vulnerability-related code regions, as well as (❷) the relationships between causal vulnerability features and prediction labels should be invariant, regardless of changes in spurious parts. Specifically, to solve the first issue, inspired by that Vulnerability-Fixing Commits (VFCs) are the most common source of current vulnerability datasets [16], Snopy adopts a change-based sample denoising approach to automatically weed out vulnerability-irrelevant code elements in the vulnerable functions without sacrificing the label accuracy. To address the second issue, Snopy constructs a Causality-Aware Graph Attention Network (CA-GAT) to approximately disentangle causal and spurious features from vulnerable samples based on attention mechanism [58], and maximize the causal effect of real vulnerability patterns on predicting labels while ignoring the spurious parts. In addition, given the diversity of spurious features in practice, CA-GAT incorporates a novel Feature Caching Scheme (FCS) to reuse previously identified spurious features without significantly raising memory consumption.

Evaluation. We implement a prototype system of Snopy, and conduct comparative experiments with nine prior DL-based vulnerability detection approaches [8, 11, 24, 36, 38, 39, 45, 61, 71] on the three well-regarded VFC-based benchmark datasets, including FFmpeg+QEMU [71], Big-Vul [20], and DiverseVul [10]. Experimental results show that Snopy outperforms the state-of-the-art baselines with respect to F1-score. In particular, compared to the best-performing baseline SVulD [45], Snopy achieves 2.35%, 13.20%, and 6.72% relative improvements on the three datasets, respectively. In addition, when applied to detect different types of vulnerabilities, Snopy produces substantial improvements of up to 21.80% in terms of F1-score on average than SVulD.

Contributions. This paper makes the following contributions:

- We propose a novel DL-based approach, Snopy[3], which bridges sample denoising with causal graph learning to capture real vulnerability patterns from vulnerable samples with numerous noise for effective detection.
- We propose a novel Causality-Aware Graph Attention Network (CA-GAT) with Feature Caching Scheme (FCS) to learn causal vulnerability features while maintaining efficiency.
- We evaluate Snopy against nine existing DL-based vulnerability detection approaches on the three large-scale benchmark datasets. The extensive experiment results indicate that Snopy outperforms the state-of-the-art baselines.

Paper Organization. The remainder of this paper is organized as follows. Section 2 introduces the background knowledge related to our problem, and discusses the limitations of existing solutions. Section 3 describes the details about our approach. Section 4 presents the experimental setup, followed by the evaluation results in Section 5. Section 6 discusses the possible threats to validity. Section 7 reviews the related work. Section 8 summarizes this paper and outlines our future research agenda.

2 BACKGROUND AND MOTIVATION

2.1 Background

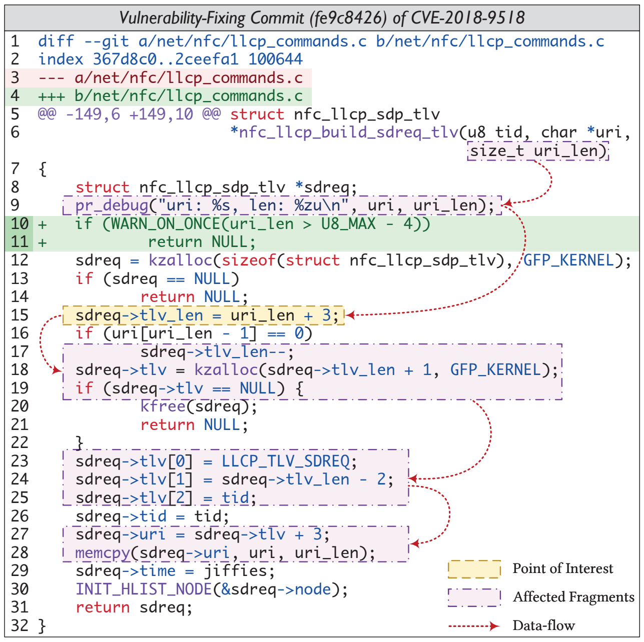
Vulnerability-Fixing Commits (VFCs), also known as *Security Patches* [34], are code changes that correct the vulnerabilities that already exist in repositories. They are commonly released to the public through official channels or vulnerability advisories, such as Common Vulnerabilities and Exposures (CVE) [13] and National Vulnerability Database (NVD) [44], for security management.

[3]https://github.com/SnopyArtifact/Snopy

$$\begin{tikzcd}
 & S \arrow[dr, dashed] & \\
F \arrow[ur, dashed] \arrow[r] & C \arrow[r] & Y
\end{tikzcd}$$

S : spurious feature
C : causal feature
Y : prediction label
F : target function

Figure 1: Structural causal model for DL-based vulnerability detection.



```
1  diff --git a/net/nfc/llcp_commands.c b/net/nfc/llcp_commands.c
2  index 367d8c0..2ceefa1 100644
3  --- a/net/nfc/llcp_commands.c
4  +++ b/net/nfc/llcp_commands.c
5  @@ -149,6 +149,10 @@ struct nfc_llcp_sdp_tlv
6                       *nfc_llcp_build_sdreq_tlv(u8 tid, char *uri,
                                                   size_t uri_len)
7  {
8     struct nfc_llcp_sdp_tlv *sdreq;
9     pr_debug("uri: %s, len: %zu\n", uri, uri_len);
10 +  if (WARN_ON_ONCE(uri_len > U8_MAX - 4))
11 +          return NULL;
12    sdreq = kzalloc(sizeof(struct nfc_llcp_sdp_tlv), GFP_KERNEL);
13    if (sdreq == NULL)
14            return NULL;
15    sdreq->tlv_len = uri_len + 3;
16    if (uri[uri_len - 1] == 0)
17            sdreq->tlv_len--;
18    sdreq->tlv = kzalloc(sdreq->tlv_len + 1, GFP_KERNEL);
19    if (sdreq->tlv == NULL) {
20            kfree(sdreq);
21            return NULL;
22    }
23    sdreq->tlv[0] = LLCP_TLV_SDREQ;
24    sdreq->tlv[1] = sdreq->tlv_len - 2;
25    sdreq->tlv[2] = tid;
26    sdreq->tid = tid;
27    sdreq->uri = sdreq->tlv + 3;
28    memcpy(sdreq->uri, uri, uri_len);
29    sdreq->time = jiffies;
30    INIT_HLIST_NODE(&sdreq->node);
31    return sdreq;
32 }
```

Figure 2: A motivating example from `Linux Kernel`.

Vulnerability-Contributing Commits (VCCs) are code changes in which the vulnerable lines of code were firstly added [42]. A straightforward way to identify VCCs is SZZ algorithm [52], which assumes that the lines of code deleted by the VFCs are the same as or evolved from the lines of code added by the VCCs.
Structural Causal Model (SCM) [49] is a directed acyclic graph that describes causal relationships between different variables. If a variable B depends on A to determine its value, the variable A is a *cause* of B and the arrow $A \longrightarrow B$ indicates the causal relationship. Figure 1 presents the causalities among four variables in the context of DL-based vulnerability detection, in which the variable C denotes the causal feature that truly reflects the intrinsic property of a vulnerability while the variable S represents the spurious feature which is usually caused by the data biases (e.g., specific coding styles). Based on the feature representation of a target function F, the neural vulnerability detector extracts the discriminative part to produce a prediction label Y (1 for vulnerable and 0 otherwise).

Scrutinizing this causal graph, we can recognize a backdoor path $F \longrightarrow S \longrightarrow Y$ between the target function F and its prediction label Y, wherein the spurious feature S plays a confounder role that establishes the statistical (but not causal) correlation which only exists in training but not in testing. As a result, the deployed detection model may suffer from serious performance degradation once the testing data distribution shifts from the original training data. To eliminate the backdoor path, a promising solution is **causal intervention** [49], which pairs each causal feature with various types of spurious features that appear in the dataset and promotes invariant relationships between causal features and predictions, regardless of changes in spurious parts.

2.2 Motivating Example

To illustrate the limitations of existing neural vulnerability detectors and motivate the key insight of our approach, we use a real-world vulnerability CVE-2018-9518[4] as a running example.

Figure 2 exemplifies the fixing commit of this vulnerability. Since the data type of `tlv_len` is `u8` (i.e., unsigned char), there exists out-of-bounds write when `uri_len` is allocated to it (line 15) without bounds checking. This issue could lead to local escalation of privilege, allowing an attacker to cause a denial of service (system crash) or possibly execute arbitrary code. To fix this vulnerability, developers perform compulsory security checks (lines 10-11) to limit the size of `uri_len` to avoid writing past the end of the allocated buffer. Benefiting from the superior capability to automatically extract features from source code, DL techniques have emerged as promising solutions for vulnerability detection [8, 24, 71]. However, as shown in this case, the core logic of a vulnerability generally involves only a few statements, while the whole function contains dozens (mostly hundreds) of lines of code. As a result, these program semantics of vulnerability-irrelevant code elements (i.e., *noise*) will heavily influence the DL model.

To mitigate the noise problem, an intuitive way is to perform program slicing [60] from vulnerability-prone program points of interest (e.g., sensitive APIs, pointer usage) to prune irrelevant statements [4, 38, 39, 63]. For example, DeepWukong [11] performed forward and backward control- and data-flow slicing based on *system API calls* and *arithmetic operators* to extract graph slices for feature extraction. As shown in Figure 2, starting from the yellow-shaded expression statement `sdreq->tlv_len = uri_len + 3` (i.e., a vulnerability-triggering statement at line 15), we can obtain a set of vulnerability-related contexts (purple-shaded) with data-dependencies (red arrow) by traversing the Program Dependence Graph (PDG) [22]. Despite their effectiveness, such pre-processing relies heavily on the expertise of the developers performing the analysis, and the knowledge (e.g., sources and sinks) of existing vulnerabilities. What's worse, according to their annotation rules (i.e., statements removed from the patched function are vulnerability-related), the code slice (yellow- and purple-shaded statements) in our motivating example will be mislabeled as *benign* because it does not contain any deletion. According to our statistical analysis on the widely-used vulnerability dataset Big-Vul [20], 24.19% vulnerable samples are fixed only with additions. This observation reveals the challenge slicing-based solutions may face in practice.

Another common approach is model-centric, which guides the DL model to autonomously learn the critical aspects (i.e., code elements of high importance) of the input sample without human

[4]https://nvd.nist.gov/vuln/detail/CVE-2018-9518

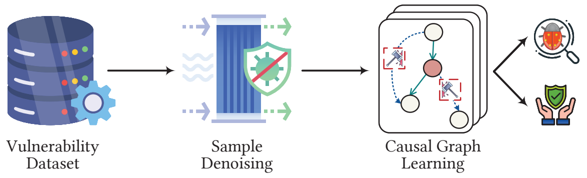


Figure 3: Workflow of Snopy.

intervention. For instance, by employing the edge-aware attention module as well as a kernel-scaled convolutional layer, AMPLE [61] estimated the importance of each edge and node and correctly identified this motivating example as vulnerable. However, recent works [7, 50] have revealed that both state-of-the-art DNN- and LLM-based approaches tend to rely on non-causal features as shortcuts to make predictions. Thus, we further calculate the node importance to locate the Top-5 important statements (an acceptable threshold on manual inspection [24, 36]) likely to be vulnerable. Unfortunately, neither the vulnerability-triggering statement (line 15) nor its corresponding contexts (purple-shaded fragments in Figure 2) are captured by AMPLE for prediction. To alleviate the problem, CausalVul [50] assumed variable and API names can be abused as spurious features, and prevented models from using them for prediction. However, crafting hundreds of variants to test various spurious features that can occur in code is not practical, and the contextual semantics of variable names (e.g., `uri_len`) are beneficial to identifying the potential buffer overflow in this case.

Our Idea. In summary, current neural vulnerability detectors (❶) struggle to focus on vulnerability-related parts code elements, and (❷) tend to exploit spurious correlations for prediction, thus undermining their effectiveness in practice. Naturally, we ask the question, *"how to capture real vulnerability patterns from vulnerable samples with numerous noise for effective detection?"* For this example, the statements added by the VFC pinpoint the vulnerability-related variable `uri_len` (line 10). Such *explicit expert knowledge* can serve as the slicing criterion that enables us to weed out vulnerability-irrelevant parts of vulnerable functions while ensuring the label accuracy. As for the spurious correlation issue, we can leverage the attention mechanism [58] to approximately disentangle causal and spurious features on representation-level, and maximize the causal effect of real vulnerability patterns on predicting labels while ignoring the spurious parts.

3 METHODOLOGY

3.1 Overview of Snopy

Figure 3 illustrates the overall workflow of Snopy. The key technical contributions of Snopy include: (❶) sample denoising, which weeds out the vulnerability-irrelevant parts of vulnerable functions for representation, and (❷) causal graph learning, which captures real vulnerability patterns for prediction. In the deployment phase, Snopy works like a typical neural vulnerability detector [8, 36, 71] to discover vulnerabilities at the function-level. We present the details of Snopy in the following subsections.

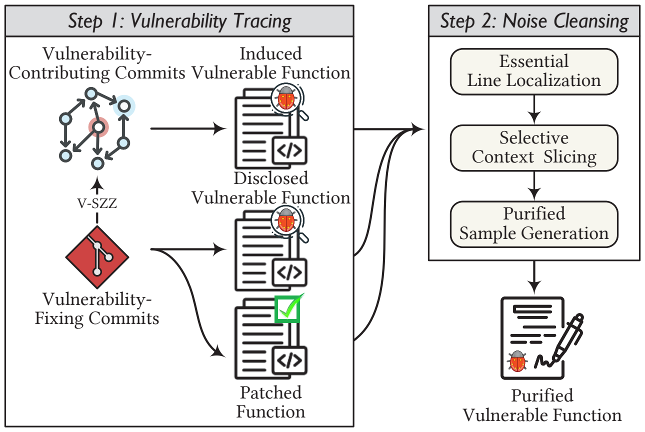

Step 1: Vulnerability Tracing
Vulnerability-Contributing Commits
Induced Vulnerable Function
V-SZZ
Disclosed Vulnerable Function
Vulnerability-Fixing Commits
Patched Function
Step 2: Noise Cleansing
Essential Line Localization
Selective Context Slicing
Purified Sample Generation
Purified Vulnerable Function

Figure 4: Overview of sample denoising.

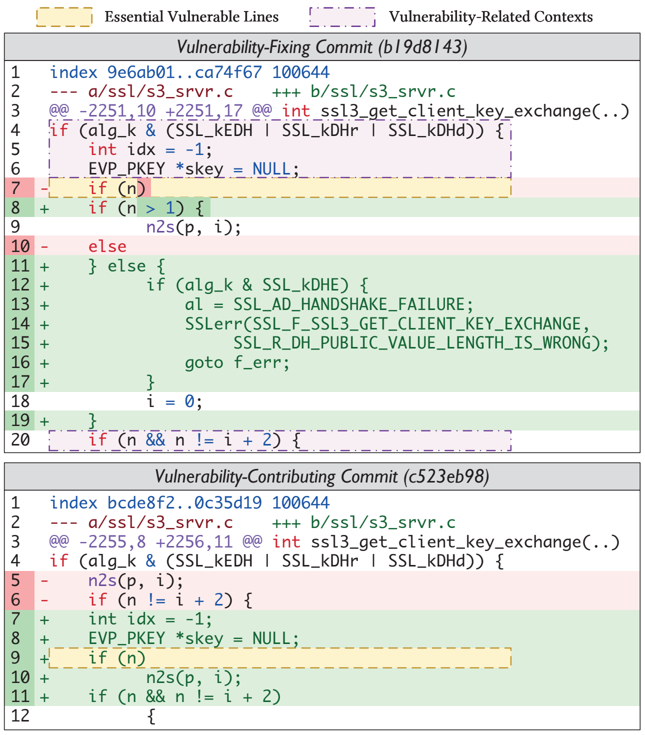
3.2 Sample Denoising

As mentioned before, feeding the entire vulnerable function into the DL model will inevitably introduce a large amount of vulnerability-irrelevant code elements that form noise, while extracting finer-grained program slices may suffer from imprecise labeling strategy and expensive expertise. Motivated by the fact that current vulnerability datasets [10, 20, 46] primarily collect vulnerable samples from Vulnerability-Fixing Commits (VFCs), we propose a change-based sample denoising strategy, which conducts noise cleansing based on the modifications between vulnerable functions and corresponding patched versions to automatically weed out the noise information in the vulnerable functions, as shown in Figure 4.

3.2.1 Vulnerability Tracing. We first execute the `git reset` command to exactly roll back the source code to the point before and after applying the patch based on the commit ID provided by the vulnerability dataset. Then, we localize all functions involving code revisions and their scopes according to the line numbers of changed code in the VFC's header lines starting with `---` and `+++` (e.g., lines 3-4 in Figure 2), and obtain the vulnerable and corresponding patched functions in pre- and post-commit versions via the `ctage`[5] parser. In addition, considering that a VFC may also contain vulnerability-irrelevant non-essential changes [32] (e.g., variable renaming, cascaded adaptations), we further leverage the state-of-the-art V-SZZ algorithm [1] to collect Vulnerability-Contributing Commits (VCCs) as references to (❶) confirm truly vulnerable statements, and (❷) narrow the search space for vulnerability-related contexts in the later noise cleansing phase. Our intuition is that vulnerability semantics persist and propagate across multiple versions (from the vulnerability introduction to the vulnerability patch) of the vulnerable functions, regardless of other internal modifications which are often benign [42]. For clarity, we refer to the vulnerable function that is identified from the VFC as the **disclosed vulnerable function**, and the vulnerable function that is identified from the VCC as the **induced vulnerable function**.

Specifically, V-SZZ first performs code similarity checking based on the edit distance to map the deleted line (annotated by the VFC)

[5]https://ctags.io/


Essential Vulnerable Lines
Vulnerability-Related Contexts
Vulnerability-Fixing Commit (b19d8143)
Vulnerability-Contributing Commit (c523eb98)

```
1   index 9e6ab01..ca74f67 100644
2   --- a/ssl/s3_srvr.c    +++ b/ssl/s3_srvr.c
3   @@ -2251,10 +2251,17 @@ int ssl3_get_client_key_exchange(..)
4   if (alg_k & (SSL_kEDH | SSL_kDHr | SSL_kDHd)) {
5       int idx = -1;
6       EVP_PKEY *skey = NULL;
7  -    if (n)
8  +    if (n > 1) {
9           n2s(p, i);
10 -    else
11 +    } else {
12 +        if (alg_k & SSL_kDHE) {
13 +            al = SSL_AD_HANDSHAKE_FAILURE;
14 +            SSLerr(SSL_F_SSL3_GET_CLIENT_KEY_EXCHANGE,
15 +                   SSL_R_DH_PUBLIC_VALUE_LENGTH_IS_WRONG);
16 +            goto f_err;
17 +        }
18           i = 0;
19 +    }
20      if (n && n != i + 2) {
```

```
1   index bcde8f2..0c35d19 100644
2   --- a/ssl/s3_srvr.c    +++ b/ssl/s3_srvr.c
3   @@ -2255,8 +2256,11 @@ int ssl3_get_client_key_exchange(..)
4   if (alg_k & (SSL_kEDH | SSL_kDHr | SSL_kDHd)) {
5  -    n2s(p, i);
6  -    if (n != i + 2) {
7  +    int idx = -1;
8  +    EVP_PKEY *skey = NULL;
9  +    if (n)
10 +        n2s(p, i);
11 +    if (n && n != i + 2)
12           {
```

Figure 5: A working example from `OpenSSL`.

in the disclosed vulnerable function to the line in the previous version. Then, it executes the `git blame` command to trace all previous commits that touched the code changes in VFCs, and considers the earliest one as the VCC. Based on these retrieved VCCs, we repeat the previously procedure to extract the induced versions of disclosed vulnerable functions. The only difference lies in that the induced vulnerable function is extracted from the post-commit version, rather than the pre-commit version.

3.2.2 *Noise Cleansing.*

Given a 3-tuple of a vulnerability, i.e., the induced vulnerable function f_i, the disclosed vulnerable function f_d, and the patched function f_p, SNOPY conducts noise cleansing to extract the vulnerability-related part. Generally, an ideal vulnerable sample should only cover (❶) essential vulnerable lines revealing how a vulnerability is caused, and (❷) contextual statements reflecting the environment where the vulnerability manifests.

Essential Line Localization. Following prior works [62, 64], essential vulnerable lines E_{vul} are statements st removed from f_p but included in f_i and f_d, and are formally defined as:

$$E_{vul} = \{st|(st \in (f_d \setminus f_p)) \wedge (st \in (f_i \cap f_d))\} \tag{1}$$

Considering that f_i and E_{vul} are not available when a vulnerability is fixed purely with additions (e.g., the motivating example in Figure 2), we further identify essential patch lines E_{patch} that are added in f_p but do not exist in f_d:

$$E_{patch} = \{st|(st \in (f_p \setminus f_d))\} \tag{2}$$

Selective Context Slicing. To precisely extract contextual statements E_{con} which have a major impact on vulnerability manifestation, we perform selective program slicing on f_d, which can be in two directions: backward and forward slicing.

- **Backward slicing** aims to localize the statements that have dependencies with essential vulnerable lines E_{vul} (or essential patch lines E_{patch} if E_{vul} is not available). Like normal backward slicing, we preserve all statements that have influence on E_{vul} ($/E_{patch}$) with respect to data- and control- dependencies.
- **Forward slicing** is to find statements affected by the vulnerability. Since taking all the subsequent statements into account will cover too many irrelevant statements [39, 64], we perform (❶) forward data-flow slicing as target statements receive variables/parameters assigned or checked by E_{vul} ($/E_{patch}$), and (❷) forward control-flow slicing only if the result of previous data-flow slicing is empty.

As a working example, let us consider CVE-2015-1787[6], an improper input validation vulnerability discovered in `OpenSSL`. Figure 5 shows its VFC (top) and VCC (bottom). Among the red-shaded statements deleted from f_d, only the yellow-shaded statement `if (n)` (i.e., line 7 in f_d and line 9 in f_i) belongs to E_{vul}. By performing the selective contextual statement slicing from E_{vul}, statements that have backward control- and data-dependencies (i.e., lines 4-6 in f_d) and forward data-dependencies (i.e., line 20 in f_d) are included in E_{con}.

Purified Sample Generation. After backward and forward program slicing, all essential vulnerable lines E_{vul} (if any) and the corresponding sliced contextual statements E_{con} are retained to generate the purified version of the disclosed vulnerable function f_d. These purified vulnerable functions will be fed along with benign samples into the DL model for training. It is noteworthy that sample denoising is only applied in the training phase because the VFC for a vulnerability is not available before it is detected.

3.3 Causal Graph Learning

To encourage the detection model to leverage real vulnerability patterns for prediction, we propose a novel Causality-Aware Graph Attention Network (CA-GAT), with detailed architecture presented in Figure 6. The CA-GAT consists of two main components: (❶) the feature extraction module for learning precise code representations; and (❷) the causal intervention module which promotes invariant relationships between causal feature and model outputs, regardless of changes in spurious parts.

3.3.1 *Feature Extraction.*

The feature extraction process is divided into two parts: embedding initialization and message propagation.

Embedding Initialization. Following [8, 61, 71], we reason about vulnerability patterns based on Code Property Graph (CPG) [65], a popular data structure which offers rich syntactic and semantic information of source code. Formally, a CPG is denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ represent the set of nodes and edges, respectively. Each node $v \in \mathcal{V}$ is initialized as a m-dimensional feature vector $\boldsymbol{h}_v^{(0)} \in \mathbb{R}^m$ by the pre-trained CodeBERT [21] model.

Message Propagation. To explicitly incorporate graph-structured contextual semantics into the representations of CPG nodes, we

[6]https://nvd.nist.gov/vuln/detail/CVE-2015-1787

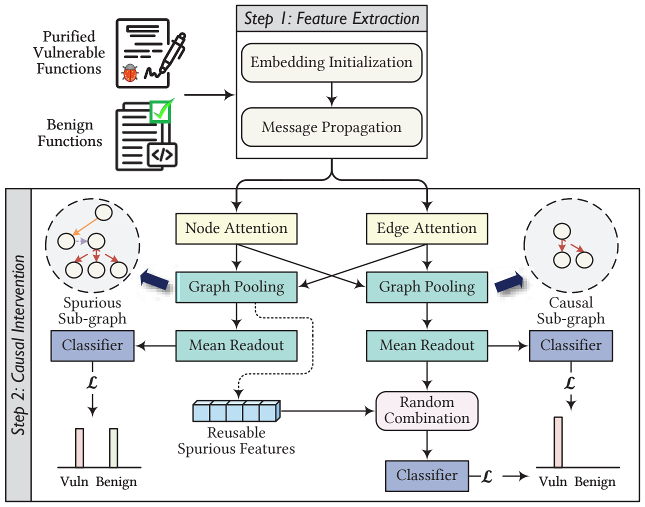


Figure 6: Illustration of our proposed Causality-Aware Graph Attention Network (CA-GAT).

employ a tailored message propagation mechanism [41] to update node features throughout graphs. To be specific, the embedding $\boldsymbol{h}_v^{(t)}$ of node v at time step $t \leq T$ is updated by:

$$\boldsymbol{h}_v^{(t)} = \text{LeakyReLU}\left(\boldsymbol{W}^{(t)}(\boldsymbol{h}_v^{(t-1)} || \boldsymbol{h}_{\mathcal{N}(v)}^{(t-1)})\right) \tag{3}$$

where $\boldsymbol{W}^{(t)}$ is a learnable transformation matrix at t-th iteration. $||$ denotes the concatenation operator. $\mathcal{N}(v)$ represents the 1-hop neighbors of node v. $\boldsymbol{h}_{\mathcal{N}(v)}^{(t-1)}$ is the neural information propagated from $\mathcal{N}(v)$ to node v at previous time step:

$$\boldsymbol{h}_{\mathcal{N}(v)}^{(t-1)} = \sum_{u \in \mathcal{N}(v)} \gamma_{u,v} \boldsymbol{h}_j^{(t-1)} \tag{4}$$

where $\gamma_{u,v}$ is a decay factor that specifies how much information is passed from node u to node v along with the edge $u \longrightarrow v$, and is formally designed as:

$$\gamma_{u,v} = \frac{1}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} \tag{5}$$

where $|\mathcal{N}(u)|$ and $|\mathcal{N}(v)|$ denote the number of 1-hop neighbors for node u and node v, respectively.

The above message propagation procedure iterates over T time steps, and the final node representation matrix is $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}| \times m}$, in which $\boldsymbol{h}_v^{(T)} = \boldsymbol{X}[v, :]$.

3.3.2 Causal Intervention. To promote the detection model learning real vulnerability patterns, we propose a novel causal intervention module with Feature Caching Scheme (FCS).

Specifically, Given an encoded CPG $\mathcal{G}$ with the adjacency matrix $\boldsymbol{A} \in \{0,1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ and node representation matrix $\boldsymbol{X}$, we first employ two Multi-Layer Perceptrons (MLP) [57] to calculate node-level and edge-level attention scores as follows:

$$\alpha_{c_v}, \alpha_{s_v} = \text{softmax}\Big(\text{MLP}_{\text{node}}(\boldsymbol{h}_v)\Big) \tag{6}$$

$$\beta_{c_{vu}}, \beta_{s_{vu}} = \text{softmax}\Big(\text{MLP}_{\text{edge}}(\boldsymbol{h}_v || \boldsymbol{h}_u)\Big) \tag{7}$$

where α_{c_v} ($/\beta_{c_{vu}}$) represents the node (/edge)-level attention score for node v (/edge $v \longrightarrow u$) in the sub-graph containing causal features (hereafter, causal sub-graph $\mathcal{G}_c$) of graph $\mathcal{G}$. Similarly, α_{s_v} and $\beta_{s_{vu}}$ are for spurious sub-graph $\mathcal{G}_s$. Note that $\alpha_{c_v} + \alpha_{s_v} = 1$, and $\beta_{c_{vu}} + \beta_{s_{vu}} = 1$.

Based on the node-level and edge-level attention scores, we can decompose the original CPG $\mathcal{G}$ into the initial causal sub-graph $\mathcal{G}_c$ and spurious sub-graph $\mathcal{G}_s$, and produce graph-level representations for them:

$$\boldsymbol{h}_{\mathcal{G}_c} = \varphi\Big(\text{GPL}(\boldsymbol{A} \odot \boldsymbol{M}_c, \boldsymbol{X} \odot \boldsymbol{F}_c)\Big) \tag{8}$$

$$\boldsymbol{h}_{\mathcal{G}_s} = \varphi\Big(\text{GPL}(\boldsymbol{A} \odot \boldsymbol{M}_s, \boldsymbol{X} \odot \boldsymbol{F}_s)\Big) \tag{9}$$

where $\varphi(\cdot)$ and GPL$(\cdot)$ represent the graph mean readout layer and pooling layer , respectively. $\odot$ denotes element-wise multiplication. $\boldsymbol{M}_c \in \{0,1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ and $\boldsymbol{F}_c \in \{0,1\}^{|\mathcal{V}| \times m}$ represent the edge mask and feature mask for the causal sub-graph $\mathcal{G}_c$. Analogously, $\boldsymbol{M}_s$ and $\boldsymbol{F}_s$ are for the spurious sub-graph $\mathcal{G}_s$.

Having obtained the representations of the causal sub-graph $\mathcal{G}_c$ and spurious sub-graph $\mathcal{G}_s$, we would like the detection model being able to disentangle them during training process for better predictions. To do so, we employ the standard cross-entropy loss over the training samples $\mathcal{D}$ to ensure the correctness of predictions based on causal features:

$$\mathcal{L}_{CE} = -\frac{1}{|\mathcal{D}|} \sum_{\mathcal{G} \in \mathcal{D}} \boldsymbol{y}_{\mathcal{G}}^{\top} \log\left(\Phi(\boldsymbol{h}_{\mathcal{G}_c})\right) \tag{10}$$

where $\Phi(\cdot)$ is the MLP-based classifier with softmax function.

For spurious features, we push their predictions evenly to binary labels (i.e., vulnerable or benign) since they are unnecessary for classification [55]:

$$\mathcal{L}_{unif} = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{G} \in \mathcal{D}} \text{KL}\left(\boldsymbol{y}_{unif}, \Phi(\boldsymbol{h}_{\mathcal{G}_s})\right) \tag{11}$$

where KL denotes the KL-Divergence. $\boldsymbol{y}_{unif}$ represents the uniform prior distribution.

As discussed in Section 2, causal intervention requires pairing the target causal feature per sample with various types of spurious features that appear in the dataset. For example, Ganz et al. [25] conditioned the causal sub-graph per sample with all possible spurious sub-graphs obtained during training. However, blindly traversing all possible spurious features (e.g., increasing the batch size) is unrealistic because any attribute of the input dataset that exhibits statistical correlation with the target label can be abused. To address this challenge, we propose a novel Feature Caching Scheme (FCS) to reuse previously identified spurious features. In each training epoch, we feed more data into the batch size (i.e., N times the batch size) and store spurious features $\boldsymbol{h}_{\mathcal{G}_{s'}}$ into the cache space $\mathcal{B} = \{\boldsymbol{h}_{\mathcal{G}_{s'}} \rightarrow \boldsymbol{b}_i\}$. In the intervention stage, we combine causal features with various spurious features in $\mathcal{B}$, and maximize the causal effect of real vulnerability patterns on predicting labels, regardless of changes in spurious parts:

$$\mathcal{L}_{cau} = -\frac{1}{|\mathcal{D}| \cdot |\mathcal{S}|} \sum_{\mathcal{G} \in \mathcal{D}} \sum_{s' \in \mathcal{S}} \boldsymbol{y}_{\mathcal{G}}^{\top} \log\left(\Phi(\boldsymbol{h}_{\mathcal{G}_c} \oplus \boldsymbol{h}_{\mathcal{G}_{s'}})\right) \tag{12}$$

where $\mathcal{S}$ is the set of reusable spurious features appearing in the training data $\mathcal{D}$. $\oplus$ denotes the random combination operation.

Table 1: Statistics of the studied datasets

Dataset	Vul	Non-vul	Ratio	VFCs	VCCs
FFmpeg+QEMU	12,460	14,858	1:1.2	6,611	6,439
Big-Vul	10,900	177,736	1:16.3	3,754	3,385
DiverseVul	18,945	311,547	1:16.4	7,514	7,022

Finally, the total training loss is defined as:

$$\mathcal{L}_{total} = \mathcal{L}_{CE} + \lambda_1 \mathcal{L}_{unif} + \lambda_2 \mathcal{L}_{cau} \tag{13}$$

where λ_1 and λ_2 are two weight coefficients indicating the strength of feature disentanglement and causal intervention, respectively.

4 EXPERIMENTS

4.1 Research Questions

Our work seeks to answer three Research Questions (RQs):

- **RQ1: How does Snopy perform compared to the state-of-the-art baselines on vulnerability detection?** By investigating this RQ, we aim to answer whether and to what extent can Snopy outperform the state-of-the-art baselines in practice.
- **RQ2: How effective is Snopy for detecting different types of vulnerabilities?** Current approaches primarily build a general-purpose model that can detect all vulnerabilities. Considering the characteristics and severity of different vulnerability types vary, we are interested to evaluate the detection performance of Snopy across different types of vulnerabilities.
- **RQ3: How do various components of Snopy affect its overall performance?** We perform two set of ablation studies to understand how different design choices, including the sample denoising (Section 3.2) and the causal graph learning (Section 3.3), impact the effectiveness of Snopy.

4.2 Datasets

We chose benchmark datasets that satisfied the following criteria. First, they should be widely adopted by state-of-the-art baselines so that we can establish a fair comparison with them. Second, they are built upon VFCs since Snopy is designed to utilize VFCs and VCCs to purify vulnerable code samples. Third, they include real-world vulnerabilities disclosed in diverse projects so that the generalization performance of approaches can be evaluated.

Based on the above criteria, we selected three popular datasets, including FFmpeg+QEMU [71], Big-Vul [20], and DiverseVul [10].

- **FFmpeg+QEMU** [71] collected security-related commits via keywords matching, and manually labeled the pre-commit function in a VFC (/non-VFC) as vulnerable (/benign).
- **Big-Vul** [20] gathered vulnerabilities from VFCs linked to CVE. Unlike FFmpeg+QEMU, Big-Vul labeled the pre-commit version of a changed function to be vulnerable, and the post-commit version to be benign.
- **DiverseVul** [10] crawled security issues to identify VFCs and extracted changed files based on relevant git commit URLs. It followed the same labeling strategy as the Big-Vul dataset.

The data statistics are shown in Table 1. Column 2 and Column 3 are the number of vulnerable and non-vulnerable functions, respectively. Column 4 denotes the ratio of vulnerable functions in each dataset. Column 5 presents the number of VFCs contained in each dataset. Column 6 reports the number of VCCs we collected from these VFCs by employing the V-SZZ algorithm. It is noteworthy that we exclude VCCs across functions (i.e., the induced vulnerable function that has an inconsistent function name as its disclosed/-patched version) since Snopy performs sample denoising based on code changes in VFCs and their corresponding VCCs.

4.3 Baselines

To evaluate our approach, we compared Snopy with nine state-of-the-art baselines, including seven DNN-based approaches [8, 11, 36, 38, 39, 61, 71] and two LLM-based approaches [24, 45].

- **DNN-based Approaches.** VulDeePecker [39] and SySeVR [38] utilized BiLSTM to extract vulnerability semantics from program slices. Devign [71], ReVeal [8], IVDetect [36], DeepWukong [11], and AMPLE [61] employed various graph neural networks to learn structural vulnerability features from code graphs.
- **LLM-based Approaches.** LineVul [24] employed a BERT architecture with self-attention layers to learn vulnerability semantics. SVulD [45] combined the strengths of pre-trained semantic embedding and contrastive learning to capture semantic representations of functions.

4.4 Experimental Setting

Implementation details. We implement Snopy in Python using PyTorch [48]. Our experiments are performed on a Linux workstation with an Intel(R) Core(TM) i9-12900k @3.90GHz, 128GB RAM, and an NVIDIA GeForce RTX 3090 GPU with 24GB memory, running Ubuntu 18.04.4 LTS with CUDA 10.1. We employ a robust parser `Joern` [65] for CPG generation. Our CA-GAT model is trained in a batch-wise fashion until convergence and the batch size is set to 128. The number of propagation iterations T is set to 4. To mitigate the overfitting problem, we employ dropout [53] with a dropping ratio of 0.1. ADAM [33] optimizer is used to train the model with a learning rate of $1e$-2. The hyper-parameters are tuned through grid search. In light of the best performance, we report experimental results in a setting with the loss coefficients λ_1 and λ_2 as 0.6 and 0.5, and the capacity N of FCS as 6 on the three datasets.
Evaluation metrics. Following prior studies [8, 39, 71], we employ four widely-used binary classification metrics, including *Accuracy*, *Precision*, *Recall*, and *F1-score*, for evaluation. Accuracy evaluates the overall performance of classifiers. Precision refers to the ratio of truly vulnerable samples among the detected ones, while Recall measures the percentage of vulnerabilities that are retrieved out of all vulnerable instances. F1-score is the harmonic mean of Recall and Precision, and calculated as: $2 \times \frac{Recall \times Precision}{Recall + Precision}$.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Detection Performance

Experiment Setup. We evaluate the nine aforementioned baselines (described in Section 4.3) and our proposed Snopy on the three studied benchmark datasets (described in Section 4.2). We randomly

Table 2: Comparison results between Snopy and state-of-the-art baselines on the three datasets

Dataset / Metrics (%)	FFmpeg+QEMU [8]				Big-Vul [20]				DiverseVul [10]			
Approach	**Accuracy**	**Precision**	**Recall**	**F1**	**Accuracy**	**Precision**	**Recall**	**F1**	**Accuracy**	**Precision**	**Recall**	**F1**
VulDeePecker	48.55	33.96	27.47	30.37	83.27	16.56	22.95	19.24	87.44	11.30	24.55	15.48
SySeVR	44.63	35.70	61.87	45.28	82.45	19.63	28.91	23.38	86.16	7.69	14.28	10.00
Devign	51.37	48.15	80.42	60.24	85.64	27.32	13.04	17.65	87.16	24.49	28.07	26.16
ReVeal	53.05	54.19	75.32	63.03	83.79	15.34	30.05	20.31	85.32	20.69	33.19	25.49
IVDetect	56.85	51.33	68.82	58.80	86.97	24.96	32.57	28.26	88.52	17.34	35.26	23.25
DeepWukong	54.61	52.70	71.96	60.84	79.64	13.08	32.55	18.66	82.39	21.64	29.30	24.89
AMPLE	62.88	55.06	77.34	64.33	85.95	28.40	36.11	31.79	88.79	26.35	34.01	29.69
LineVul	63.74	52.44	65.39	58.21	80.26	12.96	38.32	19.37	90.52	**36.18**	26.98	30.91
SVulD	60.51	54.99	**83.48**	66.30	**92.81**	33.24	41.65	36.97	**91.16**	31.44	40.17	35.27
Snopy	**67.33**	**59.64**	78.72	**67.86**	90.75	**38.12**	**46.39**	**41.85**	89.61	33.76	**42.53**	**37.64**

split the benchmark dataset into 80%, 10%, and 10% for training, validation, and testing. For each subset, we keep the distribution as same as the original ones to align with the real-world setting.

Results. Table 2 presents the performance comparisons between Snopy and state-of-the-art baselines on the three datasets regarding the four metrics (12 combination cases altogether), with the best results highlighted in bold. Overall, Snopy achieves the best performance in the majority (8 out of 12) of cases. In particular, Snopy outperforms all of the nine referred DNN-based and LLM-based approaches on the FFmpeg+QEMU, Big-Vul, and DiverseVul dataset by an average of 27.22%, 85.89%, and 75.50% in terms of F1-score, respectively. Moreover, we observe that SVulD performs better than all other baselines, although it neither focuses on vulnerability-prone code snippets like slice-level approaches [38, 39], nor explicitly utilizes the structural information of code like graph-based models [8, 11, 71]. This indicates the potential of LLM-based solutions in vulnerability detection as the size of the training set continues to expand. In contrast, our proposed Snopy consistently achieves the best performance on the three datasets in terms of F1-score, outperforming SVulD by 2.35%, 13.20%, and 6.72%, respectively. Such improvements demonstrate that Snopy strikes a better balance between reporting more vulnerabilities and reducing false positive rates.

Answer to RQ1: The performance improvements of Snopy over the state-of-the-art approaches are positive. Particularly, Snopy outperforms the best-performing baseline SVulD by 2.35%, 13.20%, and 6.72% in F1-score on the three datasets, respectively.

5.2 RQ2: Classification Performance

Experiment Setup. Common Weakness Enumeration (CWE) [14] provides a list of common software and hardware weakness types developed and maintained by security community. In particular, we focus on the Top-25 most dangerous CWEs because they are often easy to find and exploit, and should be prioritized for remediation. Here, we evaluate our Snopy and the best-performing baseline SVulD on two vulnerability datasets with CWE information, Big-Vul and DiverseVul. Minor types of vulnerabilities (i.e., sample size less than 100) are excluded to avoid the under-fitting issue due to insufficient training samples. Finally, 8 types of vulnerabilities in the Big-Vul dataset and 12 types of vulnerabilities in the DiverseVul

Table 3: The F1-score of our Snopy and baselines for the Top-25 most dangerous CWEs in Big-Vul and DiverseVul

Dataset	Rank	Type	Ratio	SVulD	Snopy
Big-Vul	1	CWE-787	2.25%	69.44	**73.89**
	4	CWE-416	3.76%	54.71	**68.32**
	6	CWE-20	13.62%	**62.99**	61.43
	7	CWE-125	7.12%	56.38	**69.47**
	12	CWE-476	2.45%	37.59	**75.23**
	14	CWE-190	3.50%	68.53	**72.06**
	17	CWE-119	24.22%	45.77	**69.28**
	21	CWE-362	3.17%	60.28	**65.33**
	Average			56.96	**69.38**
DiverseVul	1	CWE-787	17.98%	**60.47**	56.99
	4	CWE-416	6.24%	53.58	**66.29**
	6	CWE-20	8.16%	44.39	55.83
	7	CWE-125	11.60%	49.51	62.30
	8	CWE-22	1.25%	**33.65**	26.74
	12	CWE-476	6.05%	22.01	**58.14**
	13	CWE-287	0.67%	8.24	13.59
	14	CWE-190	4.86%	**61.77**	48.25
	17	CWE-119	10.14%	56.74	**64.82**
	21	CWE-362	2.84%	38.13	**55.94**
	22	CWE-269	1.22%	6.99	13.34
	23	CWE-94	0.87%	11.37	17.55
	Average			37.24	**44.98**

dataset are covered in our experiment. The first four columns of Table 3 summarize their descriptive statistics. In terms of quantity, CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) and CWE-787 (Out-of-bounds Write) have the largest proportion in the Big-Vul and DiverseVul dataset, with 24.22% and 17.98%, respectively. We build a unified model and shuffle the dataset based on vulnerability types in parallel to ensure that different types of vulnerabilities are evenly distributed in each subset. For example, 80% vulnerable samples belonging to CWE-119 are demarcated as training set, and the remaining 20% samples are divided into two halves for validation (10%) and testing (10%). We compute the F1-score to evaluate the ability of each approach to correctly predict specific types of vulnerabilities.

Results. The experimental results are presented in Table 3. Despite suboptimal performance in detecting certain types of vulnerabilities, our proposed Snopy outperforms the previous best-performing baseline SVulD in the majority of cases, with an average improvement of 21.80% and 20.78% in terms of F1-score on the Big-Vul and DiverseVul dataset, respectively. Particularly, Snopy obtains the highest F1-score of 75.23% on CWE-476 (NULL Pointer Dereference) in the Big-Vul dataset, and 66.29% on CWE-416 (Use After Free) in the DiverseVul dataset. In contrast, SVulD only achieves an F1-score of 37.59% and 53.58%, respectively. The main reason for the result is that constrained by the inherent limitations of linear Transformer-like architectures in learning structural semantics, SVulD fails to capture data-dependence information which is tightly related to CWE-416 and CWE-476. Furthermore, we observe that having more training samples for a particular CWE does not necessarily result in the model learning it better than minor types of vulnerabilities, which is consistent with a recent empirical study [54]. For instance, Snopy achieves more than 70% F1-score on CWE-787 and CWE-476, which only account for 2.25% and 2.45% of all vulnerable samples in the Big-Vul dataset. This result demonstrates the potential of Snopy in detecting certain types of vulnerabilities with limited labeled data.

Answer to RQ2: On two large-scale vulnerability datasets with CWE information, Snopy produces substantial improvements of up to 21.80% and 20.78% in terms of F1-score on average over the previous best-performing baseline in detecting different types of real-world vulnerabilities.

5.3 RQ3: Ablation Study

Experiment Setup. For sample denoising, we construct the following two variants of Snopy for comparison: (❶) without VCCs (denoted as *w/o VCCs*): performing noise cleansing only based on the disclosed vulnerable functions and corresponding patched versions in VFCs; and (❷) without sample denoising (denoted as *w/o SD*): directly feeding the entire vulnerable and benign functions into the DL model for training. For causal graph learning, we compare our proposed CA-GAT with the following four variants: (❶) without node attention (denoted as *w/o NA*): eliminating the node attention mask; (❷) without edge attention (denoted as *w/o EA*): excluding the edge attention mask;(❸) without feature caching scheme (denoted as *w/o FCS*): constraining the types of spurious features to each mini-batch; and (❹) without causal intervention (*w/o CI*): directly leveraging the graph-level embedding of the input sample for prediction like ReVeal [8]. In each ablation study, we remove one component at a time to examine the individual contributions of key designs.

Results. Figure 7 presents the results of Snopy and its six variants. We can observe that ***all key designs are essential to achieve the best performance.*** Particularly, without the sample denoising, the F1-score on the three datasets drops by 7.18%, 20.07%, and 15.59%, respectively. It demonstrates the necessity of filtering vulnerability-irrelevant parts of vulnerable samples for DL models. For causal graph learning, Snopy with the node attention, edge attention, and FCS bring 5.45%-12.77%, 2.86%-5.82%, and 5.19%-20.74% improvements in F1-score on the three datasets, respectively. Among all variants, we find that ***the causal intervention module makes the greatest contribution.*** Compared to the variant without causal intervention, Snopy improves the F1-score by 12.26%, 43.91%, and

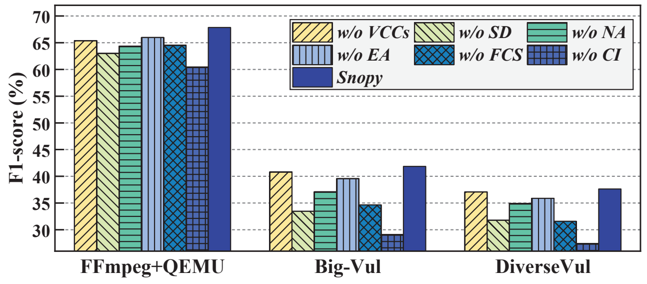


Figure 7: The performance of different variants of Snopy on the three datasets.

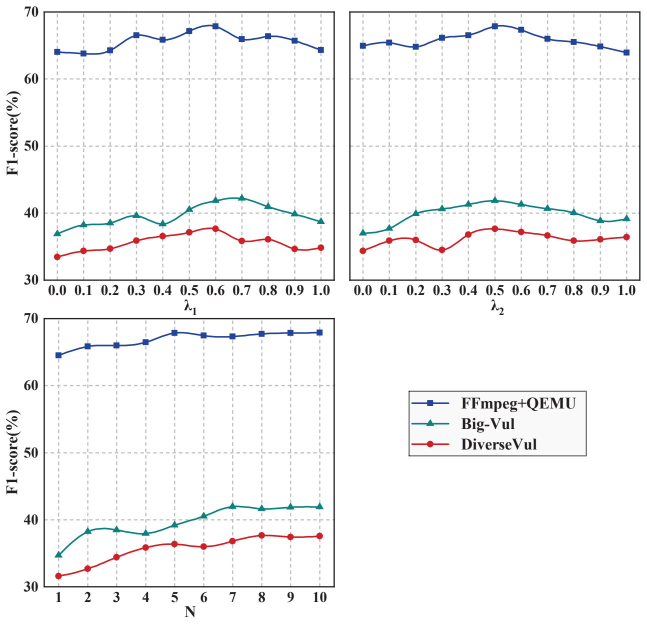


Figure 8: Parameter analysis of loss coefficients and the capacity of FCS.

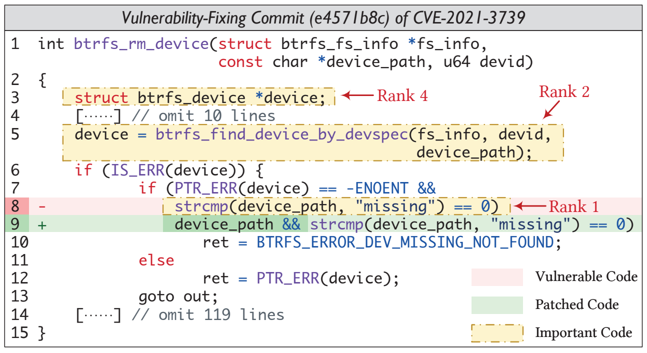
37.32% on the FFmpeg+QEMU, Big-Vul, and DiverseVul datasets, respectively. The results indicate that our proposed causal intervention module facilitates the causal feature learning and brings a performance improvement in vulnerability detection.

Answer to RQ3: Both sample denoising and causal graph learning are essential for the performance of Snopy. The most important component of Snopy is the causal intervention module that results in at most 30.51% improvement in F1-score.

6 DISCUSSION

6.1 Sensitivity Analysis

Motivation. In our approach, two sets of key hyper-parameters, loss coefficients λ_1&λ_2 and capacity N of FCS, affect the performance of Snopy. The former controls the intensity of feature disentanglement and causal intervention, the latter enriches the diversity



Vulnerability-Fixing Commit (e4571b8c) of CVE-2021-3739

```
1  int btrfs_rm_device(struct btrfs_fs_info *fs_info,
                       const char *device_path, u64 devid)
2  {
3      struct btrfs_device *device;
4      [······] // omit 10 lines
5      device = btrfs_find_device_by_devspec(fs_info, devid,
                                             device_path);
6      if (IS_ERR(device)) {
7              if (PTR_ERR(device) == -ENOENT &&
8  -               strcmp(device_path, "missing") == 0)
9  +               device_path && strcmp(device_path, "missing") == 0)
10                     ret = BTRFS_ERROR_DEV_MISSING_NOT_FOUND;
11             else
12                     ret = PTR_ERR(device);
13             goto out;
14     [······] // omit 119 lines
15 }
```

Figure 9: A NULL pointer dereference (CWE-476) vulnerability correctly detected by Snopy. The yellow-shaded statements positively contribute to Snopy's prediction.

of spurious features for causal intervention. Therefore, we vary these hyper-parameters to assess Snopy's sensitivity towards them.

Experiment Setup. To keep simplicity, all other configurations are kept consistent with RQ1, including data partition and metric computation. For loss coefficients, we fix one as 0.5, and test the other one at scale [0, 1], incrementing in steps of 0.1. For the capacity N of FCS, the value is varied from 0 to 10 with an internal of 1.

Results of Loss Coefficients. The evaluation results on the three datasets are shown in the top part of Figure 8. We can observe that, different weights of feature disentanglement λ_1 and causal intervention λ_2 have varying impact on Snopy's performance. In particular, the F1-score of Snopy stably goes up with the increasing of λ_1 and λ_2, and reaches the optimal performance around 0.6 and 0.5, respectively. After that, both two coefficients lead to varying degrees of performance degradation.

Results of Varying Capacity. Figure 8 (bottom) presents the performance of Snopy with varying capacity N. From the results, it is evident that increasing the number of candidate spurious features improves the overall performance up to a certain point, underscoring the significance of diverse spurious features in enhancing model performance. Interestingly, we find that the performance plateaus when N exceeds 6. This denotes that our Snopy can deliver satisfactory performance with limited memory overhead.

Summary: Different weights of loss coefficients have varying impact on Snopy's performance, and the larger capacity of FCS may not always guarantee better performance. Our default hyper-parameter settings achieve optimal results.

6.2 Case Study: Interpretability

Motivation. While demonstrated superior performance, the black-box nature hinders the adoption of current neural vulnerability detectors in practice because they fail to explain why a given code is predicted as vulnerable [30]. For this purpose, we conduct a case study to further investigate whether our proposed Snopy effectively makes predictions based on real vulnerability patterns.

Experiment Setup. In line with previous research [7, 36], we formalize vulnerability interpretation as a localization task, i.e., pointing out a set of crucial statements $\{s_i, \cdots, s_j\} \in P$ that are most relevant to the detected vulnerable code P. To do so, we calculate the attention score α_{c_i} of each node $v \in \mathcal{V}$ in the causal sub-graph $\mathcal{G}_c$, as defined in Equation (6). Finally, Top-5 important statements are derived as interpretations.

Results. Figure 9 presents a correctly detected vulnerability (CVE-2021-3739[7]) in the DiverseVul dataset. The vulnerable function has been simplified for a clear illustration. Since `btrfs_rm_device()` itself can have case where it only receives `devid` (at line 1), a NULL pointer dereference vulnerability can be easily triggered when calling a non-existing `device_path` via `strcmp()` (at line 8). Overall, the vulnerability-triggering statement obtains the highest attention score (i.e., Rank 1) and its contexts (line 3 and line 5), which reflect the root cause of this vulnerability, are also included in Snopy's Top-5 explanations. This proves the practicality of Snopy when applied to real-world usage because Snopy can not only accurately detect vulnerabilities, but also narrow down the scope of manual review by providing a small fragment of suspicious code elements.

6.3 Threats to Validity

Threats to Internal Validity come from the quality of our experimental datasets. We evaluated the detection performance of Snopy on the three large-scale benchmark datasets, including FFmpeg+QEMU, Big-Vul and DiverseVul. However, existing vulnerability datasets have been reported to exhibit varying degrees of quality issues such as noisy labels and duplication [15]. To reduce the likelihood of experiment biases, we adhere to the best practice [19], which discards duplicated samples by comparing the MD5 hashes of the pre- and post-commit versions of the changed functions, and labels vulnerable samples based on two customized rules.

Threats to External Validity refer to the generalizability of our approach. We only evaluate our approach on C/C++ datasets, and thus our experimental results may not generalizable to other programming languages (e.g., Java and Python). However, we believe the key unique design of Snopy (i.e., leveraging vulnerability-related code changes to reduce vulnerability-irrelevant part of vulnerable samples, and constructing CA-GAT to learn causal vulnerability features) is language-agnostic, and thus our approach can be easily ported for a new programming language. Another threat is the imprecise identification of VCCs, which may impair the accuracy of essential vulnerable line localization. To mitigate this, we employ the state-of-the-art V-SZZ algorithm to identify the induced versions from disclosed vulnerable functions.

7 RELATED WORK

7.1 Learning-based Vulnerability Detection

According to their employed model architectures, current DL-based vulnerability detectors typically can be categorized into two groups: DNN- and LLM-based approaches.

7.1.1 DNN-based Approaches. Early studies [17, 38, 39] primarily relied on sequential neural networks, such as Long Short-Term Memory (LSTM) [28] and Gated Recurrent Unit (GRU) [12], to learn vulnerability features from flatten AST or token sequence. However, these approaches failed to capture well-defined semantics of the program structure. To address this issue, Devign [71] leveraged

[7]https://nvd.nist.gov/vuln/detail/CVE-2021-3739

Gated Graph Neural Network (GGNN) [35] with a convolutional module to extract rich structural information of vulnerable code from joint code graphs. This progress promotes the recent popularity of GNN-based solutions [5, 8, 11, 36]. For instance, AMPLE [61] simplified the input program graph to alleviate the long-term dependency problems and fused local and global heterogeneous node relations for better representation learning.

7.1.2 LLM-based Approaches. As large code models have shown great potential in automating various software engineering tasks, LLMs-based vulnerability detection approaches have gained traction [18, 24, 27, 45, 67]. These works either directly train a vulnerability classifier based on fine-tuned large code models (e.g., CodeBERT [21] and UniXcoder [26]), or re-train their own LLMs from scratch to incorporate domain knowledge beneficial for downstream vulnerability analysis tasks. For example, LineVul [24] employed a BERT-like model to generate representations of source code for vulnerability detection, and leveraged the attention mechanism [68] to locate vulnerable statements. SVulD [45] adopted contrastive learning [69, 70] to fine-tune the pre-trained UniXcoder to discriminate the semantic difference among lexical similar functions.

Unlike the above approaches that adapt advanced DL models to learn vulnerability features from the entire vulnerable functions, our work bridges sample denoising with causal graph learning to suppress the negative impact of noise information while promoting learning of real vulnerability patterns for effective detection.

7.2 Causal Theory in Software Engineering

Theory of Causation [29] endows the model with the ability to pursue real causality without the interference from confounding factors. Such mechanism is a useful verification tool to achieve a more complete understanding of black-box neural models in SE tasks, such as code completion [51] and vulnerability detection [7, 47, 50]. Coca [7] proposed a dual-view causal inference-based approach to derive crucial code statements that are most decisive to the detected vulnerability as interpretations. CausalVul [50] discovered that weakly-robust variable and API names can be abused as spurious features, and applied causal learning to disable the detection model from using them for prediction. Our work is differentiated by the fact that we focus on capturing causal features for more effective vulnerability detection, instead of improving the adversarial robustness and interpretability of code models. As such, our research has a completely different goal as compared to them.

8 CONCLUSION AND FUTURE WORK

In this paper, we propose Snopy, a novel DL-based approach, which bridges sample denoising with causal graph learning for effective detection. Snopy adopts a change-based sample denoising approach to weed out vulnerability-irrelevant parts of vulnerable functions, and constructs a novel Causality-Aware Graph Attention Network (CA-GAT) to capture real vulnerability patterns for prediction. Experiments on the three public benchmark datasets show that Snopy outperforms the state-of-the-art baselines by an average of 27.22%, 85.89%, and 75.50% in terms of F1-score, respectively.

In the future, we plan to work with our industry partners to deploy Snopy in their proprietary security systems to uncover real-world vulnerabilities. In addition, we would like to develop an explanation module to provide not only fine-grained vulnerable lines, but also user-friendly explanatory information, such as the root causes, behaviors, and consequences of vulnerabilities.

ACKNOWLEDGMENTS

This research / project is supported by the National Natural Science Foundation of China (No. 62202414), the Six Talent Peaks Project in Jiangsu Province (No. RJFW-053); the Jiangsu "333" Project and Yangzhou University Top-level Talents Support Program (2022), the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University (No. KFKT2022B17), the Open Foundation of Yunnan Key Laboratory of Software Engineering (No. 2023SE201), the China Scholarship Council Foundation (No. 202308320436), and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2352–2364.

[2] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, Jiajia Li, and Tao Wei. 2023. ODDFuzz: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP)*. IEEE, 2726–2743.

[3] 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.

[4] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 1456–1468.

[5] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, Xiaoxue Wu, Chuanqi Tao, Tao Zhang, and Wei Liu. 2024. Learning to Detect Memory-related Vulnerabilities. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 43:1–43:35.

[6] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, Lili Bo, Bin Li, Rongxin Wu, Wei Liu, Biao He, Yu Ouyang, and Jiajia Li. 2023. Improving Java Deserialization Gadget Chain Mining via Overriding-Guided Object Generation. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 397–409.

[7] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, and Wei Liu. 2024. Coca: Improving and Explaining Graph Neural Network-Based Vulnerability Detection Systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 155:1–155:13.

[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning based Vulnerability Detection: Are We There Yet? *IEEE Trans. Software Eng.* 48, 9 (2022), 3280–3296.

[9] Checkmarx. 2024. https://www.checkmarx.com.

[10] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David A. Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. ACM, 654–668.

[11] 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 (2021), 38:1–38:33.

[12] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. In *Proceedings of 8th Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST)*. ACL, 103–111.

[13] Common Vulnerabilities and Exposures. 2024. https://cve.mitre.org/.

[14] Common Weakness Enumeration. 2024. https://cwe.mitre.org/index.html.

[15] Roland Croft, Muhammad Ali Babar, and M. Mehdi Kholoosi. 2023. Data Quality for Software Vulnerability Datasets. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 121–133.

[16] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. 2023. Data Preparation for Software Vulnerability Prediction: A Systematic Literature Review. *IEEE Trans. Software Eng.* 49, 3 (2023), 1044–1063.

[17] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2021. Automatic Feature Learning for Predicting Vulnerable Software Components. *IEEE Trans. Software Eng.* 47, 1 (2021), 67–85.

[18] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards Learning (Dis)-Similarity of Source Code from Program Contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 6300–6312.

[19] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David A. Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We? *arXiv preprint arXiv: 2403.18624* (2024).

[20] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*. ACM, 508–512.

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv: 2002.08155* (2020).

[22] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349.

[23] Flawfinder. 2024. http://www.dwheeler.com/FlawFinder.

[24] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 608–620.

[25] Tom Ganz, Erik Imgrund, Martin Härterich, and Konrad Rieck. 2023. PAVUDI: Patch-based Vulnerability Discovery using Machine Learning. In *Proceedings of the 39th Annual Computer Security Applications Conference (ACSAC)*. ACM, 704–717.

[26] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 7212–7225.

[27] Hazim Hanif and Sergio Maffeis. 2022. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In *Proceedings of the 31st International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[28] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780.

[29] Paul W Holland. 1986. Statistics and Causal Inference. *Journal of the American statistical Association* 81, 396 (1986), 945–960.

[30] Yutao Hu, Suyuan Wang, Wenke Li, Junru Peng, Yueming Wu, Deqing Zou, and Hai Jin. 2023. Interpreters for GNN-Based Vulnerability Detection: Are We There Yet?. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 1407–1419.

[31] Arnold Johnson, Kelley Dempsey, Ron Ross, Sarbari Gupta, Dennis Bailey, et al. 2011. Guide for Security-Focused Configuration Management of Information Systems. *NIST special publication* 800, 128 (2011), 16–16.

[32] David Kawrykow and Martin P. Robillard. 2011. Non-Essential Changes in Version Histories. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, 351–360.

[33] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

[34] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2201–2215.

[35] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*.

[36] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability Detection with Fine-Grained Interpretations. In *Proceeding of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 292–303.

[37] Zhen Li, Ruqian Zhang, Deqing Zou, Ning Wang, Yating Li, Shouhuai Xu, Chen Chen, and Hai Jin. 2023. Robin: A Novel Method to Produce Robust Interpreters for Deep Learning-Based Code Classifiers. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 27–39.

[38] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* 19, 4 (2022), 2244–2258.

[39] 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. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*.

[40] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 544–555.

[41] Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning Graph-based Code Representations for Source-level Functional Similarity Detection. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 345–357.

[42] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. 2013. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *Proceedings of the 7th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE Computer Society, 65–74.

[43] Yisroel Mirsky, George Macon, Michael D. Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. VulChecker: Graph-based Vulnerability Localization in Source Code. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. 6557–6574.

[44] National Vulnerability Database. 2024. https://nvd.nist.gov/.

[45] Chao Ni, Xin Yin, Kaiwen Yang, Dehai Zhao, Zhenchang Xing, and Xin Xia. 2023. Distinguishing Look-Alike Innocent and Vulnerable Code by Subtle Semantic Representation Learning and Explanation. In *Proceedings of the 31th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 1611–1622.

[46] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a Cross-Language Vulnerability Dataset with Commit Data. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 1565–1569.

[47] David N. Palacio, Alejandro Velasco, Nathan Cooper, Alvaro Rodriguez, Kevin Moran, and Denys Poshyvanyk. 2024. Toward a Theory of Causation for Interpreting Neural Code Models. *IEEE Trans. Software Eng.* 50, 5 (2024), 1215–1243.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems (NeurIPS)*. 8024–8035.

[49] Judea Pearl. 2000. Models, Reasoning and Inference. *Cambridge, UK: Cambridge University Press* 19, 2 (2000), 3.

[50] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2024. Towards Causal Deep Learning for Vulnerability Detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 153:1–153:11.

[51] Daniel Rodríguez-Cárdenas, David N. Palacio, Dipin Khati, Henry Burke, and Denys Poshyvanyk. 2023. Benchmarking Causal Study to Interpret Large Language Models for Source Code. In *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 329–334.

[52] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes? *ACM SIGSOFT Softw. Eng. Notes* 30, 4 (2005), 1–5.

[53] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* 15, 1 (2014), 1929–1958.

[54] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An Empirical Study of Deep Learning Models for Vulnerability Detection. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE/ACM.

[55] 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 (KDD)*. ACM, 1696–1705.

[56] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Alain Laredo, and Alessandro Morari. 2021. Probing Model Signal-Awareness via Prediction-Preserving Input Minimization. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 945–955.

[57] Bruce W Suter. 1990. The Multilayer Perceptron As An Approximation to A Bayes Optimal Discriminant Function. *IEEE Transactions on Neural Networks* 1, 4 (1990), 291.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems (NeurIPS)*. 5998–6008.

[59] 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 Trans. Inf. Forensics Secur.* 16 (2021), 1943–1958.

[60] Mark Weiser. 1984. Program Slicing. *IEEE Trans. Software Eng.* 10, 4 (1984), 352–357.

[61] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M. Zhang, and Qing Liao. 2023. Vulnerability Detection with Graph Simplification and Enhanced Graph Representation Learning. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2275–2286.

[62] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. 2022. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In *Proceedings of the 31st USENIX Security Symposium (Security)*. 3037–3053.

[63] Bozhi Wu, Shangqing Liu, Yang Xiao, Zhiming Li, Jun Sun, and Shang-Wei Lin. 2023. Learning Program Semantics for Vulnerability Detection via Vulnerability-Specific Inter-procedural Slicing. In *Proceeding of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 1371–1383.

[64] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium (Security)*. 1165–1182.

[65] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 590–604.

[66] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-trained Models of Code. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 1482–1493.

[67] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2023. Vulnerability Detection by Learning From Syntax-Based Execution Paths of Code. *IEEE Trans. Software Eng.* 49, 8 (2023), 4196–4212.

[68] Jiale Zhang, Chengcheng Zhu, Chunpeng Ge, Chuan Ma, Yanchao Zhao, Xiaobing Sun, and Bing Chen. 2024. BadCleaner: Defending Backdoor Attacks in Federated Learning via Attention-Based Multi-Teacher Distillation. *IEEE Trans. Dependable Secur. Comput.* 21, 5 (2024), 4559–4573.

[69] Jiale Zhang, Chengcheng Zhu, Xiaobing Sun, Chunpeng Ge, Bing Chen, Willy Susilo, and Shui Yu. 2024. FLPurifier: Backdoor Defense in Federated Learning via Decoupled Contrastive Training. *IEEE Trans. Inf. Forensics Secur.* 19 (2024), 4752–4766.

[70] Jiale Zhang, Chengcheng Zhu, Di Wu, Xiaobing Sun, Jianming Yong, and Guodong Long. 2021. BadFSS: Backdoor Attacks on Federated Self-Supervised Learning. In *Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI)*.

[71] 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. In *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems (NeurIPS)*. 10197–10207.