GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics

Shu Wang*§, Xinda Wang*§, Kun Sun*, Sushil Jajodia*, Haining Wang†, Qi Li‡
*George Mason University, †Virginia Tech, ‡Tsinghua University
{swang47, xwang44, ksun3, jajodia}@gmu.edu, hnw@vt.edu, qli01@tsinghua.edu.cn

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179479

***Abstract*—With the increasing popularity of open-source software, embedded vulnerabilities have been widely propagating to downstream software. Due to different maintenance policies, software vendors may silently release security patches without providing sufficient advisories (e.g., CVE). This leaves users unaware of security patches and provides attackers good chances to exploit unpatched vulnerabilities. Thus, detecting those silent security patches becomes imperative for secure software maintenance. In this paper, we propose a graph neural network based security patch detection system named GraphSPD, which represents patches as graphs with richer semantics and utilizes a patch-tailored graph model for detection. We first develop a novel graph structure called PatchCPG to represent software patches by merging two code property graphs (CPGs) for the pre-patch and post-patch source code as well as retaining the context, deleted, and added components for the patch. By applying a slicing technique, we retain the most relevant context and reduce the size of PatchCPG. Then, we develop the first end-to-end deep learning model called PatchGNN to determine if a patch is security-related directly from its graph-structured PatchCPG. PatchGNN includes a new embedding process to convert PatchCPG into a numeric format and a new multi-attributed graph convolution mechanism to adapt diverse relationships in PatchCPG. The experimental results show GraphSPD can significantly outperform the state-of-the-art approaches on security patch detection.**

I. Introduction

While the open source software (OSS) movement has made great contributions to computer software development, the number of OSS vulnerabilities also increases dramatically. As announced by the 2021 OSSRA report [1], 98% of codebases contain open source components; meanwhile, 84% of codebases have at least one open-source vulnerability and 60% of them contain high-risk vulnerabilities. By exploiting the OSS vulnerabilities reported in the vulnerability databases (e.g., NVD [2]), attackers can perform "N-day" attacks against unpatched software systems. For instance, the remote command execution vulnerability (CVE-2021-22205) was initially released on April 2021 [3]; however, after seven months, over 30,000 unpatched GitLab servers have been compromised and misused to launch DDoS attacks.

Timely software patching is an effective common practice to reduce the "N-day" attacks. Unfortunately, users or admins are often overwhelmed with the increasing large number of various patches on adding new features, resolving performance bugs, or fixing security vulnerabilities. Thus, the software updates could be postponed, due to the workflow of collecting, testing, validating, and scheduling the patches [4]. To address this software patching challenge, it becomes critical for users and admins to distinguish the security patches from other patches and prioritize the patches for fixing security vulnerabilities. However, not all the security patches are reported to NVD or explicitly recognized in the changelog. Software vendors may silently release security patches since the patch management is quite subjective [5]. For those silent security patches, it is hard for users and system admins to understand their real security impacts and hence fail to set a high priority for applying corresponding patches. Therefore, it is vital to distinguish security patches from other patches.

To identify security patches, researchers have either employed machine learning (ML) methods with syntax features [5], [6], [7] or performed recurrent neural networks (RNNs) to handle the patch code as a sequential data structure [8], [9]. However, all existing solutions have two major limitations: *lack of program semantics* and *high false-positive rate*. First, with the focus on code syntax only, they achieve a relatively low accuracy on detecting security patches. For example, the ML-based methods focus on extracting the metadata and keyword features, missing the dependencies between statements. Inspired by natural language processing (NLP) techniques, RNN-based methods segment the programs as a set of code tokens and leverage sequential models to identify security patches. However, they ignore the unique properties of programming languages on component units, dependency relationships, and token types. Second, the high false-positive rates of existing solutions limit their usages. For instance, two twin RNN based solutions [8], [9] that leverage both source code and commit messages have the false-positive rates of 11.6% and 33.2%, respectively. Considering that only 6-10% of overall patches are security-related [10], it is imperative to reduce the false positives.

In this paper, we propose a graph-based security patch detection system *GraphSPD* that consists of two core components: a graph representation of patches called *PatchCPG* and a patch-tailored graph neural network model called *PatchGNN*. PatchCPG accommodates enriched semantics information of patches by extending the code property graph (CPG) [11] to integrate both pre-patch and post-patch versions of the source code. After converting the graph topology of PatchCPG into a numeric format with patch-tailored features, PatchGNN uses a new multi-attributed graph convolution mechanism to adapt diverse relationships in PatchCPG.

We first develop PatchCPG, a new intermediate graph representation, to embed richer syntax and semantic information of patches. Several graph representations have been developed for

§The first two authors contributed equally to this work.

DOI 10.1109/SP46215.2023.00035

analyzing source code [12], [13], [14], [15], [16], [17], [18], [19]; however, none of them are dedicated for representing the code changes between two different versions. By extending the code property graph [11], PatchCPG is a directed edge-labeled and attributed multigraph, where the nodes store attributes of statements and the edges are labeled with different types of relations between the nodes. Particularly, it contains new node attributes and edge labels to mark the added, deleted, or context code for a given patch.

The basic principle of constructing a PatchCPG is to retain the relevant identical context components and accommodate the added and deleted nodes/edges in a unified graph. To this end, we first retrieve both the pre-patch and post-patch files of the source code and remove irrelevant functions. Then, after generating two CPGs for both pre-patch and post-patch functions, we merge these two CPGs into a joint structure. Further, a program slicing technique is adopted to retain the most relevant components to the changed code.

Using PatchCPG as input, our graph neural network based model PatchGNN can detect if the patch is a security or non-security patch. Since the graph representation cannot be directly processed by deep learning models, PatchCPG is first embedded into a numeric format, where the graph topology is denoted by an adjacent matrix and the edge/node attributes are represented by embedding vectors. We customize 20 patch-related features for each statement as the node embedding. The edge embedding is a vector representing the version information and edge types. To process the diverse edge attributes in PatchCPG, we propose a new multi-attributed convolution mechanism that views each dimension of edge embeddings as an individual convolution channel. The graph convolution operates in each subgraph individually with different weights; and then the multi-attributed convolution result is obtained by aggregating the information from all subgraphs.

We implement a prototype of GraphSPD on detecting C/C++ security patches with 5K LoC in Scala and Python. Then, we conduct extensive experiments over two public C/C++ patch datasets: PatchDB [10] and SPI-DB [9]. The experimental results indicate that GraphSPD can achieve up to 80.4% detection accuracy, with a relatively low false-positive rate of 5%. Our further analysis reveals that the most valuable context in PatchCPG is carried by the statements directly connected with the changed ones.

We conduct performance comparison with two categories of detection mechanisms. First, we directly compare our work with other security patch detection methods. Compared with the RNN based methods [8], [9], our system can achieve much higher accuracy along with a significant reduction of the false-positive rate (the false-negative rate remains the same). Second, vulnerability detection tools can be leveraged to identify security patches since they should detect vulnerabilities in the pre-patch version and detect no such vulnerabilities in the post-patch version. Compared with five state-of-the-art vulnerability detection approaches [20], [21], [22], [23], [24], GraphSPD can boost the detection rate by at least 2.5 times.

We also evaluate the real-world performance of GraphSPD by conducting a case study to detect the silent security patches on four popular open source repositories, i.e., NGINX, Xen, OpenSSL, and ImageMagick. Among 137 identified patches by GraphSPD, 88 are security patches that are not assigned with CVE IDs and/or explicitly described in the changelog or the commit messages.

In summary, we make the following contributions:

- We present a security patch detection system GraphSPD that consists of two key components: PatchCPG and PatchGNN, achieving high accuracy and fewer false alarms.
- We develop a novel graph data structure PatchCPG for patches by leveraging the rich semantics of both pre-patch and post-patch source code. To the best of our knowledge, PatchCPG is the first graph-based patch representation that can significantly improve detection accuracy.
- We propose a graph learning model PatchGNN to detect security patches. PatchGNN utilizes multi-attributed convolution to adapt the diverse relationships in PatchCPGs and adopts patch-tailored features to reduce false alarms.
- We implement a prototype of GraphSPD and conduct experiments to evaluate its effectiveness and efficiency.[1] We also perform case studies on four open source repositories to validate the practicality of GraphSPD on real-world projects.

II. Preliminaries

A. Security and Non-Security Patches

Open source software patches record the changes between two different versions of source code. The best practices for Git commit include making single-purpose commit [25], [26], and we observe it is usually followed by well-maintained Git repositories. In our work, we define a "patch" as a single-purpose Git commit for addressing a security vulnerability (a security fix), resolving a functionality bug (a non-security repair), or updating a new feature. Among them, security patches are usually considered more urgent and given higher priority than non-security ones. We consider a patch as a "security patch" if it fixes a vulnerability belonging to any Common Weakness Enumeration Specification (CWE) type [27], no matter what trigger condition it may apply to.

Listings 1 and 2 exemplify a security patch and a non-security patch, respectively. The code revision is represented as a set of consecutive deleted and/or added statements (i.e., the lines starting with a single − or +). In Listing 1, an `if` statement (Line 8) is added before the original assignment statement to check if the pointer `tcon` is valid. This security patch mitigates a NULL pointer dereference vulnerability, where an invalid pointer with a value of NULL can lead to a crash or exit. Listing 2 shows an example of non-security patch, where an obsolete identifier `hack` is deleted since it is no longer used in the subsequent code, but this variable definition (Line 7) will not incur any security problems.

[1] Our study is performed on OSS patches that can be publicly accessed on NVD and GitHub. To help admins and developers identify silent security patches and prioritize their deployment, we release our source code at https://sunlab-gmu.github.io/GraphSPD.

```
commit 18f39e7be0121317550d03e267e3ebd4dbfbb3ce
diff --git a/fs/cifs/smb2pdu.c b/fs/cifs/smb2pdu.c
@@ -907,7 +907,8 @@ tcon_exit:
 tcon_error_exit:
     if (rsp->hdr.Status == STATUS_BAD_NETWORK_NAME) {
         cifs_dbg(VFS, "BAD_NETWORK_NAME: %s\n", tree);
-        tcon->bad_network_name = true;
+        if (tcon)
+            tcon->bad_network_name = true;
     }
     goto tcon_exit;
 }
```

Listing 1: An example of security patch (CVE-2018-19200).

```
commit 6cf575b1ad989fbb8a239dd6acc26d72286eb4cb
diff --git a/src/path.c b/src/path.c
@@ -1145,7 +1145,6 @@ int git_path_diriter_init(
     unsigned int flags)
 {
     git_win32_path path_filter;
-    git_buf hack = {0};

     static int is_win7_or_later = -1;
     if (is_win7_or_later < 0)
```

Listing 2: An example of non-security patch.

By default, patches contain 6 neighboring code lines (e.g., Line 4-6 and 8-10 in Listing 2) as the context for each code revision. However, these context statements may not provide sufficient semantics to understand the patches. Fortunately, to perform code analysis on an OSS patch, we can retrieve the source code before and after applying this patch, and we call them *pre-patch* and *post-patch* files/functions, respectively. A patch is considered as a security patch when a vulnerability exists in the pre-patch code and the corresponding fix statements are in the post-patch code. Usually, security patches can involve sanity checks, which are security checks on critical values like bound, permission, etc. For example, for the patches to fix use-after-free and double-free vulnerabilities, sanity checks are the added conditional statements to check the availability of pointers or memory.

B. Code Property Graph

Code property graph (CPG) [11] is a language-agnostic intermediate program representation, which merges multiple abstract representations of source code into one queryable graph database. The CPG merges three compiler representations (i.e., Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG)) into a single joint data structure. AST is a code representation generated by the syntax analysis of a compiler. CFG is a graph structure that represents all the possible traversed paths during program execution. PDG comprises of control dependency graph (CDG) and data dependency graph (DDG) to represent the control and data dependencies, respectively [28]. By containing all the information of control flow, control dependency, intra-procedural data dependency, and program syntax, CPG provides a comprehensive view for code static analysis.

Multiple CPG-assisted approaches [14], [15], [18], [29] are developed for vulnerability detection. The open-source platform *Joern* can generate CPGs and represent the output CPGs with nodes, labeled directed edges, and key-value pairs (i.e., node attributes) [30]. We extend the CPGs for patches with more semantics between pre-patch and post-patch code.

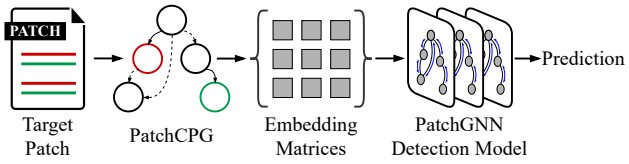


Fig. 1: Workflow of GraphSPD system.

III. System Overview

A. System Model and Workflow

The overall workflow of GraphSPD is illustrated in Figure 1. The inputs of GraphSPD are OSS patches, which are first mapped to an intermediate graph representation PatchCPG that contains control/data dependency and program syntax. Next, the nodes and edges of PatchCPG are converted into embedding matrices, which are fed into a graph-based deep neural network PatchGNN that predicts if the input patches are either security or non-security patches.

Mapping Patches to PatchCPGs. Before using the deep learning techniques to identify security patches, we need to first convert the input patches into a proper type of intermediate representation that can be processed by neural networks. Compared with the code sequential representations (e.g., processed by the NLP techniques), the graph representations can embed richer information such as control/data dependency. Therefore, we develop a novel graph-based data structure called PatchCPG to represent the patches in C/C++.

Due to the limited context information in the patch files themselves, we first retrieve the pre-patch and post-patch source code to obtain more context details. For a security patch, the pre-patch source code can reveal the vulnerability patterns, while the post-patch source code can indicate the fixing details. Therefore, we build two CPGs of the pre-patch and post-patch source code, respectively. PatchCPG is a data structure constructed by merging the CPGs of pre-patch and post-patch source code. The merging principle is to retain the shared context components in both CPGs and then attach the deleted and added components from the pre-patch CPG and post-patch CPG, respectively. Therefore, PatchCPG is a unified graph containing nodes and edges from two different versions.

However, introducing more context from the source code will also bring a large amount of noise (i.e., irrelevant context information) into the detection process, thus impairing the system performance. We mitigate this problem by removing irrelevant context from two aspects. First, before the CPG extraction from source code, all functions that are not involved in the patch are removed from the source code files. Second, after merging CPGs, a program slicing technique is applied over PatchCPG to limit the range of context code according to the hop count towards the nodes of deleted/added statements.

Detecting Security Patches via PatchGNN. To identify security patches, PatchCPG instances will be first transformed into a numeric format and then fed into the PatchGNN. To embed a PatchCPG into a numeric graph G, we convert the topology into an adjacency matrix and embed the attributes of edges and nodes. The edges in PatchCPG are embedded into 5-dimensional vectors using the corresponding version

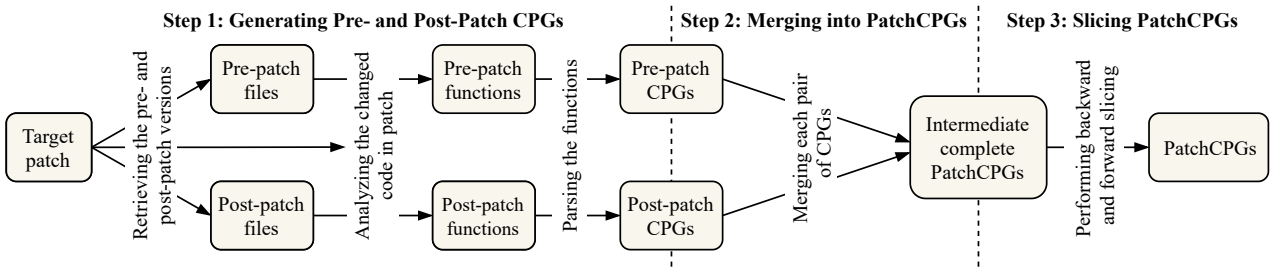


Fig. 2: Overview of PatchCPG construction.

information and edge types. The nodes in PatchCPG are embedded into 20-dimensional vulnerability-relevant features, which are extracted from the node attributes and involved statements. These features are customized for patches, so they are crucial for reducing false alarms in security patch detection. Node embeddings contain 2 code-independent features (i.e., code snippet metadata) and 18 code-dependent features (i.e., features of identifiers, literals, control flows, operators, and APIs). To extract the code-dependent features, the statement in each node is segmented as a set of code tokens with different token types. The feature extraction is based on token (or sub-token) matching and token type recognition.

The architecture of PatchGNN is based on graph convolution networks, which learn the model via message propagation along the neighboring nodes. Since PatchCPG contains multiple attributes (e.g., version information, edge types), we propose a multi-attributed convolution mechanism to achieve convolution operations in different subgraphs and aggregate the information from all subgraphs. Here, the PatchGNN is a classification model that can be described formally as $f_p : G(V, E) \mapsto [0, 1]^2$. The detector output is a vector (p_0, p_1) representing two class probabilities ($p_0 + p_1 = 1$). The training objective is to find the optimized parameters of f_p to minimize the cross-entropy, i.e., $\min_{f_p} \sum -(y \log(p_1) + (1-y) \log(p_0))$, where y denotes a binary indicator (0 or 1) showing if the input is a real non-security or security patch. A patch is determined as the category with a higher possibility in detection phase.

B. Assumption

We assume that the source code of both pre-patch and post-patch versions can be accessed via version control systems (e.g., Git). Except the patches and the involved source code, we do not require any descriptive documentations (e.g., commit messages and changelogs) since the quality of such documentations highly relies on their maintainers and some commit messages are not accurate or even empty. Our GraphSPD system focuses on the patches in the C/C++ language, which contain more high-severity vulnerabilities than other programming languages. In the past 10 years, 52.13% of the reported vulnerabilities in open source software are written in C/C++ [31]. Our high-level design guidance for graph-based security patch detection can be adapted to other languages.

IV. Constructing Patch Code Property Graphs

As illustrated in Figure 2, a PatchCPG can be constructed in three steps, namely, generating CPGs, merging CPGs into a PatchCPG, and reducing the size of PatchCPG.

A. Generating Pre-Patch and Post-Patch CPGs

Since a patch contains pre-patch and post-patch code, we can generate pre-patch and post-patch CPGs, respectively.

Retrieving Pre-Patch and Post-Patch Source Code. Though a patch contains multiple lines of context code (e.g., three lines ahead and behind the changed code snippet by default), we may miss some critical context out of this range. To solve this problem, we retrieve the files of full source code before and after applying the patch. Each patch on the Git repositories can be uniquely identified by a commit ID (i.e., a 20-byte SHA-1 hash). Given the commit ID of a specific patch, the source code of the corresponding Git repository can be rolled back exactly to the point before and after applying this patch by using the `git reset` command. Thus, we can obtain the source code files of both pre-patch and post-patch versions.

Identifying Patch Related Functions. There may be multiple code files in each software version; however, we only focus on the files modified by the patch. These files can be identified by the header lines starting with `---` and `+++` (e.g., Line 3-4 in Listing 3). In these files, we focus on the functions containing code revisions. To include global variables in our analysis, we remove unrevised functions from the related files instead of retaining all revised functions. We first identify all functions and their scopes (i.e., the line number range between function start and function end) via Joern parser [32]. A patch contains the range information showing the line numbers of changed code in pre-patch and post-patch files, e.g., in Line 5 of Listing 3, Line 3439 is deleted from the unpatched file and Line 3444-3447 are added to the patched file. We compare their scopes with those revised by the patch. After removing irrelevant functions, we obtain the patch-related functions.

Constructing Pre-Patch and Post-Patch CPGs. Given the patch-related functions in pre-patch and post-patch versions, we apply the Joern parser [32] to generate two CPGs (i.e., G_{pre} and G_{post}). Given a CPG, we describe the graph with two sets: (V, E). V is a set of nodes represented with 2-tuple $(id, code)$, where id is a number to identify the node and $code$ is the source code component depicted by this node, i.e., a code token in AST or a statement in CDG/DDG. E is a set of directed edges represented with 3-tuple $(id_1, id_2, type)$, where id_1 and id_2 represent the IDs of start and end nodes. $type \in \{AST, CDG, DDG\}$ is the edge type indicating if the edge belongs to the AST or denotes control/data dependency. Note that Joern also provides CFG, but we do not include it since the information of CFG has been included by CDG.

```
1  diff --git a/core/tee/tee_svc_cryp.c b/core/tee/
       tee_svc_cryp.c
2  index a5beb339cc9..f60f3d248c2 100644
3  --- a/core/tee/tee_svc_cryp.c
4  +++ b/core/tee/tee_svc_cryp.c
5  @@ -3439 +3444,4 @@ TEE_Result syscall_asymm_verify(
       unsigned
6  TEE_Result syscall_asymm_verify(unsigned long state,
       const struct utee_attribute *usr_params, size_t
       num_params, const void *data, size_t data_len,
       const void *sig, size_t sig_len)
7  {
8      TEE_Result res;
9      TEE_Attribute *params = NULL;
10     struct user_ta_ctx *utc;
11
12     ...
13
14     res = tee_mmu_check_access_rights(utc,
        TEE_MEMORY_ACCESS_READ |
        TEE_MEMORY_ACCESS_ANY_OWNER, (uaddr_t)sig, sig_len)
        ;
15     if (res != TEE_SUCCESS)
16         return res;
17
18 -   params = malloc(sizeof(TEE_Attribute) * num_params);
19 +   size_t alloc_size = 0;
20 +   if (MUL_OVERFLOW(sizeof(TEE_Attribute), num_params,
       &alloc_size))
21 +       return TEE_ERROR_OVERFLOW;
22 +   params = malloc(alloc_size);
23
24     if (!params)
25         return TEE_ERROR_OUT_OF_MEMORY;
26     res = copy_in_attrs(utc, usr_params, num_params,
       params);
27     if (res != TEE_SUCCESS)
28         goto out;
29
30     ...
31
32 out:
33     free(params);
34     return res;
35 }
```

Listing 3: A security patch for buffer overflow vulnerability (CVE-2019-1010298).

B. Merging into PatchCPG

For each pair of pre-patch and post-patch functions, we merge the corresponding two CPGs into a unified graph structure called PatchCPG. The function names are used to pair the functions in the pre-patch files with the corresponding ones in the post-patch files. According to the code revision of patches, we define three types of components in a PatchCPG.

- **Deleted components**. They are nodes and edges in the pre-patch graph and do not appear in the post-patch one. For a security patch, the deleted components are highly relevant to the vulnerabilities.
- **Added components**. They are nodes and edges that only exist in the post-patch graph and are not in the pre-patch one. For a security patch, the added components are usually the operations to fix the vulnerabilities.
- **Context components**. They are the nodes and edges corresponding to unchanged statements, which appear in both pre-patch and post-patch functions. Though these components are not modified by patches, they contain context information that are related to the deleted or added statements.

The main idea of merging into PatchCPG is to retain the context components and attach the deleted and added components in a unified graph. Given the pre-patch and post-patch CPGs, the algorithm will merge them into a PatchCPG in three steps. *(1) Identifying the node versions.* For each node, we identify if it is a deleted, added, or context component. *(2) Identifying the edge versions.* For each edge, we will identify if one of its connected nodes is deleted or added components. If so, the edge is marked as the corresponding component. An edge is a context component only if both of its connected nodes are context. *(3) Re-assigning node IDs and merging the node and edge sets.* We re-assign each node a new node ID since the node IDs in pre-patch and post-patch CPGs may be conflicting. After updating node IDs in the edge sets, we merge the node/edge sets of pre-patch and post-patch CPGs into a unified node/edge set. In this way, we obtain a PatchCPG depicted by two sets (V', E'), where $V' = V_{pre} \cup V_{post}$ and $E' = E_{pre} \cup E_{post}$. Then, we append an additional element $version$ to each tuple of nodes/edges. Thus, the node set V' is represented with 3-tuples $(id, code, version)$ and the edge set E' is represented with 4-tuples $(id_1, id_2, type, version)$, where $version \in \{deleted, added, context\}$ is the version information denoting which type of component in PatchCPG the node/edge belongs to.

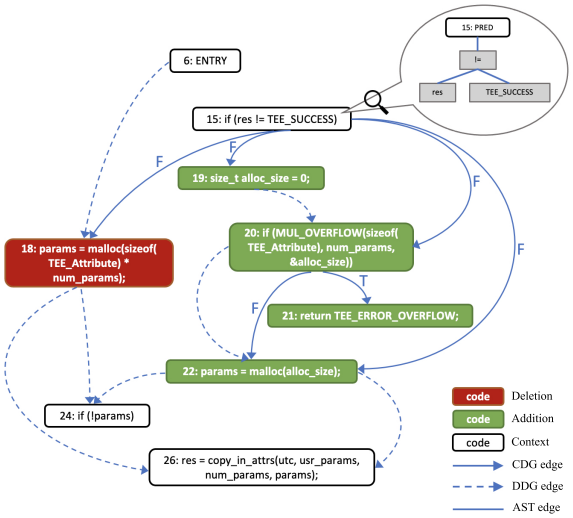


Fig. 3: Patch Code Property Graph (PatchCPG) of the patch in Listing 3 (the edge version and most AST parts are omitted).

C. Computing Slices over Code Changes

In PatchCPG, it is unnecessary to include all the statements in a function since some of them are irrelevant to vulnerabilities. To locate the vulnerability-relevant context statements, we perform program slicing [33] on the source code with the criterion of deleted/added statements. Different from the traditional definition in a patch where adjacent statements of changed code are regarded as context [34], we define our sliced statements as context since only these statements have dependencies with the changed ones. Our slicing can be performed in two directions: backward and forward slicing.

- **Backward slicing** is to locate the source of the vulnerability. For example, when we set a deleted statement (e.g., Line 18 in Listing 3) as the criterion, the results of back slicing are the statements in Line 6 and 15. Line 18 is data dependent

on Line 6 since the variable `params` is determined by the argument `num_params` in the current function. Line 18 is control dependent on Line 15 since Line 18 can be executed only when the condition in Line 15 is not satisfied. Otherwise, the function will directly return. After backward slicing, the nodes of Line 6 and 15, the data dependency edge between Line 6 and 18, as well as the control dependency edge between Line 15 and 18 will be retained as backward context.

- **Forward slicing** is to find statements affected by the vulnerability. For example, in Listing 3, when we set the added statement in Line 22 as a criterion, the results of forward slicing are the statements in Line 24, 26, and 27 that are directly/indirectly data dependent on the variable `params`. However, the forward slicing results of Line 20 include all the subsequent statements, i.e., Line 21-34. That is because the program will directly return in Line 21 if the condition in Line 20 is true. In this case, considering all the subsequent statements as context will lead to too much noise since they are not highly relevant to the criterion statements. Thus, when the criterion happens to be a conditional statement that leads to a function exit point (e.g., *return*), we no longer consider the subsequent slicing results with control dependency. Therefore, the nodes of Line 24, 26, and 27 as well as the data dependency edges between Line 22 and them are retained as forward context.

After performing backward and forward slicing using control and data dependency, we retain all the nodes of changed and sliced statements, as well as the traced edges. Note that the slicing is only conducted in the CDG and DDG where each node represents a statement. In PatchCPG, each AST is constructed based on a statement node (i.e., a node in CDG or DDG). Therefore, after determining the nodes in CDG and DDG, it is trivial to include all the AST components that are dependent on the retained context nodes. We can iteratively conduct slicing to find all the context statements that directly and indirectly depend on the criterion statements. In Listing 3, Line 24 and 26 is directly dependent on the variable `params` in Line 22. Since `params` decides the value of `res` in Line 26 while Line 27 checks if `res` is equal to a specific value, Line 27 is indirectly data dependent on Line 22. Moreover, more statements in the omitted part is indirectly dependent on Line 22 and they are less relevant to the changed statement. To reduce the noisy portion, we can empirically set the number of iteration times, denoted as N. For instance, when we set $N = 1$, the sliced statements of the changed code in Listing 3 will include Line 6, 15, 24, and 26. When $N = 2$, we will further add Line 27 into the PatchCPG. We will discuss more about the settings of N in our experiments (Section VI-D).

Figure 3 presents the constructed PatchCPG for Listing 3, where the solid/dotted edges denote control/data dependencies and red/green/white rectangles represent deleted/added/context statements. Here, we set $N = 1$ so that all the retained context statements are directly dependent on the changed code. In Figure 3, we omit most AST subgraphs of each statement and only show an example for statement of Line 15.

V. Graph Learning for Security Patch Detection

A. Embedding Methods

To feed PatchCPGs into a GNN-based model, the attributes in the graphs should be embedded into numeric vectors. The graph attributes contain two parts: edge attributes and node attributes. Edge attributes represent the relationships between the nodes in the graphs. Node attributes represent the code snippet in each node, either a statement or a code token.

Edge Embedding. Edge embedding is used to reflect the relationships between two nodes. The edges in PatchCPGs involve two types of relationships, i.e., version information and edge types. The version information refers to whether the edge is present only in the pre-/post-patch version or in both versions. The edge type refers to whether the edge belongs to the CDG, DDG, or AST. It is possible that there are two edges (one from CDG and the other from DDG) between the two nodes. Therefore, the edge embedding is designed as a 5-dimensional binary vector. The first two bits are used to indicate if the edge is present in the pre-patch version and the post-patch version, respectively. If the edge belongs to both versions, the first two bits will be (1, 1). The last three bits indicate if there are any CDG, DDG, or AST edge between current two nodes, respectively.

Node Embedding. Node embedding is a numeric representation of the code in each PatchCPG node, which can be a statement in CDG/DDG or a code token in AST. In this paper, the content in a node is called a *code snippet* no matter it is a statement or a token. The node embedding is a 20-dimensional vector describing the attributes of involved code snippet. Comments are not included in code snippets due to the various coding styles and vague reference scope. The C/C++ code snippets are first segmented into code tokens via *clang* tool. Then, these tokens are identified into 4 types: keywords (e.g., `if`), identifiers (variable and function names), literals (strings and numbers), and punctuation (e.g., ++). Finally, we extract the features highly related to security patch detection.

PatchGNN is able to learn the vulnerability patterns from both the syntax-level and semantic-level representations. The semantic representation is exhibited by the PatchCPG structure with diverse edge relationships, while the syntax-level representation is achieved by the node embeddings of code snippets. Recent studies [35], [7] show source code vulnerabilities have a high correlation with some specific syntax characteristics. For instance, the pointer and array usages have a higher possibility to be vulnerable in C/C++ language since these operations usually lead to the out-of-bounds (OOB) access or NULL pointer dereference. Moreover, several specific arithmetic expressions can indicate potential improper operations (e.g., integer overflow). Based on the observed syntax characteristics of vulnerabilities, the following 20 features belonging to 5 groups are extracted from the code snippet of each node.

- **Code Snippet Metadata** (2): the number of characters, the version information (i.e., deleted, added, or context).
- **Identifier and Literal Features** (7): the number of function calls, variables, numeric numbers, strings, pointers, arrays, and null identifiers.

TABLE I: The involved tokens or sub-tokens of the control flow features, the operator features, and the API features.

Features	Matched Tokens or Sub-tokens
condition	if, switch
loop	for, while
jump	return, break, continue, goto, throw, assert
arithmetic†	`++`, `--`, `+`, `-`, `*`, `/`, `%`, `=`, `+=`, `-=`, `*=`, `/=`, `%=`
relational	`==`, `!=`, `>=`, `<=`, `>`, `<`
logical	`&&`, `||`, `!`, and, or, not
bitwise⋆	`&`, `|`, `<<`, `>>`, ∼, ∧, bitand, bitor, oxr
memory API	alloc, free, mem, copy, new, open, close, delete, create, release, sizeof, remove, clear, dequene, enquene, detach, attach
string API	str, string
lock API	lock, mutex, spin
system API	init, register, disable, enable, put, get, up, down, inc, dec, add, sub, set, map, stop, start, prepare, suspend, resume, connect,

† Operator `*` is determined as dereference operator or arithmetic operator.
⋆ Operator `&` is determined as address-of operator or bitwise operator.

- **Control Flow Features** (3): the boolean features indicating if the node is a conditional, loop, or jump statement.
- **Operator Features** (4): the number of arithmetic, relational, logical, and bitwise operators.
- **API Features** (4): the boolean features indicating if the code snippet contains the APIs of memory operations, string operations, lock operations, and system operations.

The node embedding is a numeric vector composed of the above 20 code features. The metadata features are directly derived, while the identifier and literal features are based on the *clang* tokenization and the identified token types. The control flow features and operator features are determined by the exact matching of token keywords, as listed in Table I. Since the API names are defined by developers, we provide a set of sub-tokens based on our observation (as shown in the last four rows in Table I). If a function name contains one of these sub-tokens, the corresponding API feature is enabled. Also, the sub-token matching scheme is case insensitive.

B. PatchGNN Model

General Design. PatchGNN is based on graph learning, which provides a great capability of graph classification by neural networks. Due to the diverse edge attributes, PatchCPG is a heterogeneous graph. To better leverage the hidden knowledge within PatchCPG, we construct the PatchGNN model with a multi-attributed graph convolution mechanism. Figure 4 shows the architecture of PatchGNN model. After embedded into a numeric format, the PatchCPG instances are first fed into three multi-attributed graph convolutional layers. The graph convolutional layers will update the node embeddings of PatchCPG with the neighborhood information in different subgraphs. We only use 3 convolutional layers in PatchGNN because more convolution will lead to graph over-smoothing. After the 3-layer graph convolution, graph embeddings can be obtained by the graph pooling and vector concatenation. Graph embedding is a type of graph representation where all the nodes, edges, and their features are transformed into a unified vector. Finally, a binary predictor constructed by multiple layer perceptron (MLP) is utilized to convert the graph embeddings

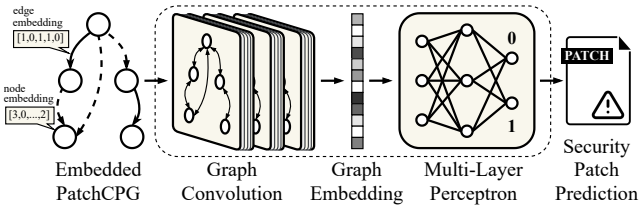


Fig. 4: The architecture of PatchGNN model.

into predicted labels. PatchGNN is the first end-to-end deep learning model that determines if a patch is security-related directly from its graph-structured information.

Multi-attributed Graph Convolution. For each node in PatchCPG, the convolutional layers in PatchGNN tend to gather information from its neighbors via different types of edges. Due to the different roles of edge types, we cannot use a set of unified weights to learn the detection model. Each dimension of the edge embeddings can indicate different relationships in PatchCPG, i.e., pre-patch/post-patch connections, control/data dependencies, and AST graph. As shown in Figure 5, each edge can have multiple attributes while each type of attributes can be used to construct a subgraph. As the information in each dimension corresponds to a subgraph, we can view each dimension as an individual convolution channel. In the convolutional layers of PatchGNN, we process the structural information of each channel individually and aggregate the processed information of all subgraphs.

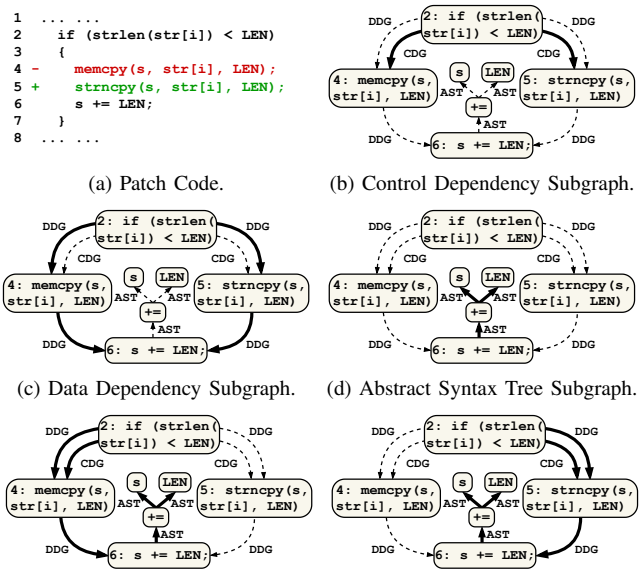
The matrix of edge embeddings is denoted as $\boldsymbol{E}$, where $E_d^{(k)}$ is the k-th attribute of the d-th edge. $\boldsymbol{E}^{(k)}$ is a vector contains the k-th attribute of all edges. Because $E_d^{(k)}$ can be either 0 or 1, we generate a masked adjacency matrix $\boldsymbol{M}^{(k)}$ according to $\boldsymbol{E}^{(k)}$, where $M_{ij}^{(k)} = 1$ if the edge connecting node i and node j has the k-th attribute of 1, else $M_{ij}^{(k)} = 0$. $\boldsymbol{M}^{(k)}$ can be used to reflect the node connections of the k-th subgraph. The multi-attributed graph convolution can be formulated as

$$\boldsymbol{X}^{(h+1)} = \frac{1}{K}\sum_{k=1}^{K}\sigma((\boldsymbol{A}\odot\boldsymbol{M}^{(k)} + \boldsymbol{I}_N)\cdot\boldsymbol{X}^{(h)}\cdot\boldsymbol{W}_h^{(k)}), \quad (1)$$

where $\boldsymbol{A}$ is the adjacency matrix of PatchCPG, $\boldsymbol{I}_N$ is the identity matrix of size N, and $\odot$ is the Hadamard product. $\boldsymbol{X}^{(h)}$ is the node embeddings in the h-th convolution layer. $\boldsymbol{W}_h^{(k)}$ is the convolution weights of the k-th subgraph in the h-th layer, which is obtained by graph model learning. K is the total number of subgraphs and σ is the activation function.

As illustrated in Figure 6, the structural information from different subgraphs will be aggregated to the convolution result $\boldsymbol{X}^{(h+1)}$ by the multi-attributed graph convolution. The masked adjacency matrix $\boldsymbol{M}^{(k)}$ can be thought of as a filter that provides individual attention for each subgraph to learn convolution parameters.

Security Patch Predictor. Since the output of convolutional layers are graphs, we need further processing to obtain the final predictions. First, the graph pooling layers are leveraged to reduce the data dimension and acquire the graph embeddings. In our design, we use both mean pooling and max pooling to obtain two graph representations, each of which is a high-



(a) Patch Code. (b) Control Dependency Subgraph.

(c) Data Dependency Subgraph. (d) Abstract Syntax Tree Subgraph.

(e) Pre-patch Subgraph. (f) Post-patch Subgraph.

Fig. 5: The subgraphs of PatchCPG where each edge can have multiple attributes and each attribute can be converted as an individual convolution channel in PatchGNN (the solid/dotted edges represent node connection/disconnection in subgraphs).

dimensional vector. Then, these two graph representations are concatenated to construct a final graph embedding, which contains all information of the nodes, edges, and their features in a PatchCPG. Afterwards, a dropout layer is performed as a regularization method to prevent over-fitting in the model training. To determine if a patch is security-related, a 3-layer perceptron is built to transform the graph embedding into a 2-unit softmax output (p_0, p_1), where $p_0 + p_1 = 1$. Each unit in the output indicates the probability that a patch instance falls into the category of non-security/security patch.

VI. Experimental Evaluation

A. Implementation

To retrieve the related files in pre-patch and post-patch versions, we implement a parser to analyze the patch, which is not a complete program unit. Since the Joern [32] can only produce separate CPGs for pre-patch or post-patch code, we extend it using Scala scripts to perform the graph merging and PatchCPG generation. For program slicing, we develop Python scripts to analyze control/data dependency and AST information and output a sliced PatchCPG ready to be embedded.

We achieve the embedding methods by Python. Specifically, we utilize the *clang 6.0.0* tool for C/C++ code tokenization and token type identification so that we can embed the nodes in PatchCPG for graph learning. The implementation of PatchGNN is built on the deep learning library *PyTorch 1.6* that is optimized for tensor computing. We develop and optimize our graph model based on the *pytorch-geometric 1.6* library, which supports deep learning on graphs and other structured data [36]. In total, we construct our GraphSPD system with 5K LoC in Scala and Python.

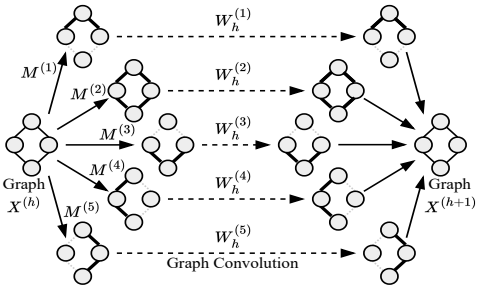


Fig. 6: The multi-attributed graph convolution mechanism in PatchGNN designed for the diverse relationships of PatchCPG.

B. Experimental Setup

Runtime Environments. All the experiments are conducted on a Ubuntu 20.04 server with a Intel Xeon 5122 CPU running at 3.60GHz, 512 GB RAM, and 2 NVIDIA GeForce RTX 2080 Ti GPU cards. The deep learning architecture is built on the NVIDIA CUDA Toolkit 11.2 and cuDNN v8.1.

Datasets. We select two benchmark datasets: PatchDB [10] and SPI-DB [9]. PatchDB is a security patch dataset that contains 12K security patches and 24K non-security patches in the C/C++ language. It comprises a NVD-based dataset crawled from the NVD reference hyperlinks and a wild-based dataset collected from GitHub commits. The samples in PatchDB are derived from 311 open source repositories (e.g., *Linux kernel*, *MySQL*, *OpenSSL*, *Vim*, *Wireshark*, *httpd*, *QEMU*, etc.), providing the chance to test the cross-project performance of security patch detection. SPI-DB is a security patch dataset originally collected from *Linux*, *FFmpeg*, *Wireshark*, and *QEMU*, including 38,291 patches. Only a portion of the dataset on *FFmpeg* and *QEMU* was released, containing 25,790 patches (10K security patches and 15K non-security patches). Overall, the above two datasets provide us with sufficient patch variants for evaluating cross-project and intra-project performance. Also, the sample ratio can balance the real-world utility and the model fitting.

Model. The PatchGNN model contains three multi-attributed convolutional layers, each reducing the dimension of node embeddings to 1/2 of the original one. In addition, the first layer aggregates subgraph information with concatenation, while the last two layers use the mean aggregation the same as Equation (1). The number of subgraphs (K) is 5, including pre-patch/post-patch graph, control/data dependency graph, and AST graph. Therefore, the dimension of node embeddings after the 1st, 2nd, and 3th convolution is 50, 25, and 12, respectively. For the predictor, the dropout rate is set to 0.5 during the model training. The dimension of hidden layers in the final multi-layer perceptron is (24, 8, 2). For the model learning, 80% of randomly selected samples are used to train the model parameters and the remaining 20% samples are used for testing. This ratio holds for both security patches and non-security patches. Also, we do not separate the training/test set by the repository. The batch size is set to 128 and the learning rate is 0.01. More details can be found in Appendix A.

State-of-the-Art Approaches. We conduct a comparison with two categories of the state-of-art approaches. First, we directly

compare our work with other security patch detection methods. We choose the RNN-based solutions [8], [9] that utilize a twin RNN scheme to determine if the patch is security-related. Specifically, two RNN modules with shared weights are deployed to process the pre-patch and post-patch code sequences, respectively. We reproduce an RNN-based model named TwinRNN according to these two works. Since commit messages highly rely on maintenance policies and some commit messages are not accurate or even empty, the TwinRNN model is implemented only leveraging the source code revision, which meets practical needs in the real world and provides a fair comparison with GraphSPD. We verify the TwinRNN performance is consistent with that in the original papers if we use both commit message and code revision.

Since few works utilize source code to identify security patches directly, we consider leveraging existing vulnerability detection tools to assist security patch detection. Given the pre-patch and post-patch functions, we assume these tools can identify security patches only if they can detect the vulnerabilities in the pre-patch function but cannot detect those vulnerabilities in the post-patch version. We select three common baseline approaches including Cppcheck [20], flawfinder [21], and ReDeBug [22] as well as two other most effective works in this field, i.e., VUDDY [23] and VulDeePecker [24]. The source codes of Devign [19] and MVP [37] were not ready to be released yet. Among them, Cppcheck and flawfinder are rule-based vulnerability detection tools, ReDeBug and VUDDY are clone-based methods, and VulDeePecker is a learning-based approach.

Evaluation Metrics. GraphSPD can be regarded as a classification model, so we use both general metrics and special metrics to evaluate its effectiveness and practicality. General metrics, including accuracy and F1-score, are used to evaluate the overall performance of the classification model. Special metrics are used to evaluate the practicability of the detection system, including precision and false-positive rate (FPR).

C. Performance Evaluation

1) Overall Results: As illustrated in Table II, the GraphSPD system can achieve up to 80.39% accuracy with a F1-score of 0.557 on PatchDB. On SPI-DB, the accuracy of GraphSPD is 63.04% with 0.503 F1-score. Note the performances on PatchDB and SPI-DB are not comparable because the data distributions are different in these two datasets. PatchDB is collected based on the similarity with NVD samples. To construct SPI-DB, raw patches are pre-screened with security-related keywords in commit messages. Then the filtered samples are labeled manually. We adopt both datasets as baseline to compare our work with existing solutions in next subsection.

2) Comparison with Security Patch Detection Approaches: We compare GraphSPD with TwinRNN over both PatchDB and SPI-DB datasets by applying the same training and test set splitting. The experimental results are summarized in Table II.

Effectiveness. Our GraphSPD system outperforms TwinRNN with 10.8% higher accuracy and 0.096 higher F1-score on PatchDB. On the SPI-DB, the GraphSPD can outperform TwinRNN with 6.67% higher accuracy with a minuscule drop in F1-score (less than 0.01). Compared with the previous work, the main difference of the GraphSPD system is to capture more enriched syntax and semantics via graph-based patch representation and patch-tailored feature extraction, thus the GraphSPD is more effective than other sequential models.

TABLE II: The performance comparison of GraphSPD and RNN-based sequential model for security patch detection.

Method	Dataset	General Metrics		Special Metrics	
		Accuracy	F1-score	Precision	FP Rate
TwinRNN [8]	PatchDB	69.60%	0.461	48.45%	19.67%
	SPI-DB	56.37%	0.512	49.07%	41.57%
GraphSPD	PatchDB	80.39%	0.557	77.27%	5.05%
	SPI-DB	63.04%	0.503	63.96%	19.16%

Practicality. To effectively reduce the update frequency and increase labor efficiency, precision and false positive rate are important metrics. When performing GraphSPD on PatchDB, Table II shows 77.27% of predicted security patches are real security-related, and only 5.05% of real non-security patches are misidentified. However, when using TwinRNN, only 48.45% of predicted security patches are real, incurring low efficiency in the practical applications. Due to the extreme imbalance between non-security and security patches in OSS (security ones only account for 6-10%), false positives matter more than false negatives. Therefore, our method aims to reduce FPR while keeping the same level of FNR. The false-positive rate in our system is only a quarter of that in other RNN-based systems, with the same level of false-negative rate (56.51% for GraphSPD and 55.95% for TwinRNN). That means in practice the number of false alarms reported by TwinRNN is above four times more than that reported by GraphSPD, while the number of detected real security patches is the same. On SPI-DB, GraphSPD outperforms TwinRNN in precision by 14.89 percentage points; the FPR of GraphSPD is only half that of TwinRNN.

3) Comparison with Vulnerability Detection Approaches: Due to the fact that few works use code revision to directly identify security patches, we consider utilizing state-of-the-art vulnerability detection tools to identify security patches since they are supposed to detect vulnerabilities in the vulnerable version and cannot detect such vulnerabilities in the patched version. We apply five effective techniques including Cppcheck, flawfinder, ReDeBug, VUDDY, and VulDeePecker on a dataset including 368 security patches of known CVEs from 5 projects. We retrieve the pre-patch (vulnerable) and post-patch (patched) code of related functions to see if these tools can detect the security patches, i.e., detect vulnerabilities in pre-patch code without incurring any false positives in post-patch code. Note that we do not include non-security patches in such a dataset since it is laborious and time-consuming to check if the pre-patch and post-patch code contain any potential vulnerabilities.

For VUDDY and VulDeePecker, we directly use their provided fingerprint dictionary or training dataset to train the model. Since ReDeBug does not provide an individual dataset

TABLE III: Comparison results with existing vulnerability detection methods on 368 security patches.

Method	# $Vul_{pre\text{-}patch}$	# $Vul_{post\text{-}patch}$	# $Patch_{security}$	TP Rate
Cppcheck [20]	3	1	2	0.54%
flawfinder [21]	109	108	1	0.27%
ReDeBug [22]	29	29	0	0.00%
VUDDY [23]	22	16	21	5.71%
VulDeePecker [24]	3	0	3	0.82%
GraphSPD	-	-	53	14.40%

for template generation, we employ the same PatchDB dataset to train both ReDeBug and our PatchGNN. These training samples have no overlapping with above 368 security patches.

Table III shows the number of vulnerabilities detected in pre-patch and post-patch code, respectively. For a security patch, a detector may falsely report its patched version as vulnerable, unpatched version as invulnerable, or even both versions as vulnerable or invulnerable. Cppcheck detects 3 vulnerabilities in pre-patch code and 1 out of them is detected as vulnerable in post-patch code, which means Cppcheck only detects 2 security patches. Flawfinder reports 109 and 108 vulnerabilities in pre-patch and post-patch code, respectively. Among them, the overlapping 108 vulnerabilities are detected in both versions of code. Recall that a security patch can only be determined if its pre-patch version is vulnerable while its post-patch version is invulnerable. Therefore, only one security patch can be successfully detected by Flawfinder. ReDeBug detects 29 vulnerabilities in both pre-patch and post-patch code, which means no security patches can be detected. VUDDY detects 22 vulnerabilities in pre-patch code, where 21 out of them are correctly detected as secure in the post-patch version. Thus, 21 security patches are finally identified. Note that these 16 vulnerabilities detected in the post-patch code do not correspond to the same patches as the previous 22 vulnerabilities. VulDeePecker only identifies 3 and 0 vulnerabilities in the pre-patch and post-patch code, respectively, so it can only detect 3 security patches.

In contrast, GraphSPD detects 53 out of 368 security patches, which outperforms the true positive rate of Cppcheck, flawfinder, ReDeBug, VUDDY, and VulDeePecker by 13.58, 14.13, 14.40, 8.69, and 13.58 percentage points, respectively. We conclude that vulnerability detection approaches can be applied to detect security patches, but their performances are not good for practical use. It shows the value of our GraphSPD model that is dedicated to security patch detection.

4) Case Analysis: To learn how GraphSPD outperforms other approaches, we study patch cases only detected by GraphSPD and exemplify three scenarios (S1-S3) as follows.

S1: pre-patch code has misleading secure patterns. As shown in Listing 4, the pre-patch code already checks `current_frame` (Line 2) before operating on it (Line 6). However, this patch (CVE-2011-3934) fixes a double free by changing the field length (i.e., replace `current_frame` with `keyframe`). Rule-based methods cannot detect it since its pre-patch version looks secure, but actually it is vulnerable. As a learning-based method, GraphSPD is capable of detecting it since similar samples have already been included in the training dataset, as shown in Listing 5.

```
1  commit 247d30a7dba6684ccce4508424f35fd58465e535
2  if (!s1->current_frame.data[0]
3      ||s->width != s1->width
4      ||s->height!= s1->height) {
5      if (s != s1)
6  -       copy_fields(s, s1, golden_frame, current_frame);
7  +       copy_fields(s, s1, golden_frame, keyframe);
8      return -1;
9  }
```

Listing 4: Security patch for a double free (CVE-2011-3934).

```
1  commit 360e95d45ac4123255a4c796db96337f332160ad
2  if (priv->cac_id_len) {
3      serial->len=MIN(priv->cac_id_len, SC_MAX_SERIALNR);
4  -   memcpy(serial->val,priv->cac_id,priv->cac_id_len);
5  +   memcpy(serial->val,priv->cac_id,serial->len);
6      SC_RETURN(card->ctx,SC_DEBUG_NORMAL,SC_SUCCESS);
7  }
```

Listing 5: A patch with similar patterns (CVE-2018-16393).

S2: patches involve complex control flow changes. The security patch in Listing 6 fixes an uninitialized `cred` pointer and determines control flow via `cred`. However, rule-based methods cannot detect it since `cred` is not present in pre-patch function (i.e., lack of key information to detect uninitialized use). Moreover, since it is difficult to generalize rule-based methods in complex control flow changes, it is challenging to summarize a general rule for this case.

```
1  commit 3440625d78711bee41a84cf29c3d8c579b522666
2      if (IS_ERR(bprm.file))
3          return res;
4  +   bprm.cred = prepare_exec_creds();
5  +   res = -ENOMEM;
6  +   if (!bprm.cred)
7  +       goto out;
8      res = prepare_binprm(&bprm);
9      if (res <= (unsigned long)-4096)
10         res = load_flat_file(&bprm, libs, id, NULL);
11 -   if (bprm.file) {
12 -       allow_write_access(bprm.file);
13 -       fput(bprm.file);
14 -       bprm.file = NULL;
15 -   }
16 +   abort_creds(bprm.cred);
17 +out:
18 +   allow_write_access(bprm.file);
19 +   fput(bprm.file);
20     return(res);
```

Listing 6: Security patch for uninit use (CVE-2009-2768).

S3: unique patterns. The patch in Listing 7 shows an uncommon pattern for fixing a double free. Instead of deleting free statements, developers can also guarantee the memory does not get freed before the release. Since `usbtv` will be freed twice via `usbtv_video_free()` and `kfree()`, this patch increments the reference count of usb device structure to avoid double free. Such a pattern is hard to be described in the studied rule-based methods.

```
1  commit 50e7044535537b2a54c7ab798cd34c7f6d900bd2
2  usbtv_audio_fail:
3  +   /* we must not free at this point */
4  +   usb_get_dev(usbtv->udev);
5      usbtv_video_free(usbtv);
6  usbtv_video_fail:
7      usb_set_intfdata(intf, NULL);
8      usb_put_dev(usbtv->udev);
9      kfree(usbtv);
```

Listing 7: The security patch for a double free on Linux kernel.

TABLE IV: The performance comparison of GraphSPD with different scope of context (slicing iteration number N).

Slicing Iteration No. (N)	General Metrics		Special Metrics	
	Accuracy	F1-score	Precision	FP Rate
0	80.19%	0.555	76.11%	5.42%
1	80.39%	0.557	77.27%	5.05%
2	79.24%	0.531	73.61%	5.87%
∞	76.90%	0.501	64.42%	8.95%

D. Performance with Different Context Scope

Context statements can assist the detection of security patches by involving more dependencies and semantics. However, as stated in Section IV, too much context can introduce interfering noise to the detection model. Therefore, the scope of context will directly affect the detection performance. We conduct experiments to evaluate the impact of different context scope on system performance. The context scope is represented by the iteration number of program slicing with the added/deleted statements as criterion, which is denoted as N. In each iteration, we extract the statements that have control/-data dependency with current criterion statements. We evaluate the system performance on the constructed PatchCPGs with four different settings:

(a) $N = 0$: only changed statements (i.e., no slicing).
(b) $N = 1$: changed statements and direct context statements.
(c) $N = 2$: changed statements along with direct and nearest indirect context statements.
(d) $N = \infty$: changed statements along with all direct and indirect context statements.

Table IV shows that GraphSPD can achieve the best performance when only considering direct context statements ($N = 1$). Compared with no context ($N = 0$), direct context can facilitate the security patch detection by complementing more semantic information. The performance gets worse when N is greater than 1, since indirect context may introduce too much noise and provide limited valuable information. The accuracy drops by 1.15% when $N = 2$, indicating that the indirect context does not carry much relevance for the changed code. The accuracy further drops by 3.49% when considering all the indirect context ($N = \infty$), because most of these context statements are barely correlated with the changed statements but introduce excessive noise that significantly interferes with the detection model.

Our experiments prove that program slicing is a straightforward and effective method to control the context scope, compared with other methods (e.g., the weighting scheme in Appendix B). The direct context provides the most valuable information for inference, and the indirect context has limited relevance to the changed statements but introduces noise.

E. Performance over Different Types of Patches

As a large-scale real-world dataset, PatchDB provides us the possibility of evaluating our system performance over different patch types, which are manually labelled according to the types of resolved vulnerabilities. Table V illustrates the performance of GraphSPD for different patch types, which are ranked with the severity of corresponding vulnerabilities [38].

TABLE V: The performance of GraphSPD over patches with different vulnerability types.

Severity	Vulnerability Type of Patch	Proportion	TP Rate
1	buffer overflow	18%	30.9%
2	improper input validation	2%	5.8%
3	resource leakage	22%	71.4%
4	double free/use after free	5%	51.1%
5	integer overflow	5%	37.2%
6	NULL pointer dereference	12%	66.0%
7	improper authentication	4%	14.7%
8	uncontrolled resource consumption	1%	33.3%
9	race condition	7%	56.6%
10	uninitialized use	8%	27.5%
n/a	other vulnerabilities	16%	16.7%

We also list the proportion for each patch type, as well as the corresponding true-positive rate (TPR) of GraphSPD. TPR, i.e., recall, refers the percentage of correctly detected samples among security patches. The overall recall of GraphSPD is 43.5%, which is the same as that of RNN models.

By analyzing the performance over each type of security patches, we have two key findings. First, for security patch types with higher TPR, the corresponding patches *exhibit distinguishable features* from non-security ones, shedding light on the detection system design. For instance, resource leakage is usually fixed with memory reinitialization and file operations, hence associated with memory APIs. Moreover, the race condition fixing always utilize the lock/unlock operations to restrict processes/threads, thus the patches are related to the lock APIs. More detailed examples are shown in Appendix C.

Second, the security patch types with low TPR are usually associated with *security check modifications*, e.g., improper input validation, buffer overflow, and improper authentication, which usually use `if` statements to restrict the operating range. Although security check is a typical pattern for security patches, sometimes it is easy to get mixed up with non-security patterns since developers also like to use conditional statements to add new features in specific cases. Seizing clues from context becomes even more decisive in these cases. Also, data imbalance affects detection over these security patches; for instance, uncontrolled resource consumption only counts for 1% in training set, providing insufficient patterns for deep learning. This effect will be mitigated if we have more data.

F. Overhead Evaluation

Illustrated in Appendix D, 80% of PatchCPGs in our experiments have fewer than 100 nodes and fewer than 200 edges, with an appropriate graph complexity. The PatchGNN model takes 42 *min* for training with 400 epochs, occupying 3 GB RAM with the CPU usage of 32% and the GPU usage of 56%. For detecting security patches, PatchGNN takes a mean of 8 *ms* for testing a single PatchCPG input.

The overhead of GraphSPD is mainly on constructing PatchCPGs for given patch files. To evaluate the scalability of PatchCPG construction, we apply our system to the commits of Linux kernel, OpenSSL, and Xen. In particular, we use all GitHub commits between version 5.17-rc1 and 5.17-rc2 of Linux, 1.1.0i to 1.1.1, 1.1.1 to 1.1.1a, and 1.1.1a to 1.1.1b of OpenSSL, and 4.15.0 to 4.15.1 of Xen. The size of these repositories varies from 88MB to 3.5GB and number of commits

TABLE VI: Overhead of PatchCPG generation.

Repo and Version	Size	#Commits	#Functions	#LoC	$\overline{T}^{\star}$
Linux: 5.17rc1-5.17rc2	3.5GB	452	618	6,788	31.7s
OpenSSL: 1.1.1a-1.1.1b	250MB	182	398	12,438	62.3s
OpenSSL: 1.1.1-1.1.1a	250MB	168	241	6,023	41.8s
OpenSSL: 1.1.0i-1.1.1	250MB	4,959	6,709	709,460	39.6s
Xen: 4.15.0-4.15.1	88MB	101	206	3,870	37.4s

* $\overline{T}$ means the average generation time per commit.

between these neighboring versions is from 168 to 4,959. The fourth and fifth columns present the total number of revised functions and revised lines of code (LoC) for all commits. Based on the implementation of PatchCPG construction, the time of PatchCPG generation is less likely related to the size of whole repository. It mostly depends on the number of revised lines and relevant context lines for a given patch. Therefore, as shown in Table VI, the average time to generate a PatchCPG is from 31.7s to 62.3s, which is acceptable in practice.

VII. Case Study

Besides large-scale datasets like PatchDB and SPI-DB, we further perform a case study on four other popular OSS to evaluate the adaptability of our system. We select NGINX (a web server software), Xen (a hypervisor project), OpenSSL (a TLS/SSL and crypto library), and ImageMagick (an image processing tool). Note that all samples used in this case study are not included in the training set. The detection outputs of GraphSPD are manually checked by three experienced security researchers, who cross-check their analysis results. They identify a security patch if it fixes a vulnerability belonging to any CWE types. We list all security patches detected by GraphSPD as well as our analysis results in Appendix E.

NGINX. We collect commits between neighboring major versions from NGINX's GitHub repository and input them into GraphSPD after filtering out invalid commits that do not contain source code changes. As presented in Table VII, there are 180 commits between NGINX 1.19.0 and 1.21.0 (two neighboring mainline versions). Our system retains 127 valid commits that have code changes. After being transformed into PatchCPGs and fed into PatchGNN model, 7 cases are detected as potential security patches. We manually confirm 6 cases as real security patches, while the NGINX changelog only shows 3 cases are reported in the CVE. Also, we perform detection on the 1.17, 1.15, and 1.13 series. Overall, GraphSPD detects 27 potential security patches from 787 input commits and 21 out of 27 are confirmed as real security patches. The detection precision is 78% (only 22% false alarms), which is consistent with the performance on benchmark datasets like PatchDB, showing the considerable generalization ability of GraphSPD.

Xen. We fetch the latest 1,170 commits (as of 11/09/2021) from the GitHub repository of Xen and input them into our system. The prediction results show that 29 commits are detected as security patches. After manually checking, 16 out of them are real security patches (i.e., 55% in precision).

OpenSSL. We test GraphSPD on the newest 1,000 commits (as of 03/11/2022) of OpenSSL GitHub repository. GraphSPD labels 68 commits as security patches and 45 out of them are real security patches after verification, i.e., 66% in precision.

TABLE VII: Security patch detection results on NGINX.

Changes w/	CVE	Total Commits	Valid Commits	Detected S.P.	Confirmed S.P.	Precision
1.19.x	3	180	127	7	6	86%
1.17.x	3	134	82	4	3	75%
1.15.x	1	203	120	7	4	57%
1.13.x	1	270	157	9	8	89%
Sum.	8	787	486	27	21	78%

S.P. = security patches

ImageMagick. Similarly, we retrieve the newest 1,000 commits (as of 03/28/2022) of ImageMagick GitHub repository. 13 commits are detected as security patches by the GraphSPD and we manually verify that 6 out of them are real security patches (i.e., 46.2% in precision).

Analysis on False Positives. GraphSPD is designed to ensure a low false positive rate. To this end, we manually check the false positives of our system and find they mainly fall into two categories. First, GraphSPD mislabels some non-security patches that replace the inefficient or outdated function calls, which are similar to security patches that replace insecure C/C++ functions. Such cases are hard to be distinguished since our current design does not consider the semantics of the callee functions. The second type is about the cross-function modification, e.g., moving statements from one function to another. Indeed, GraphSPD can handle patches that modify multiple functions since we simply build an individual PatchCPG for each function and feed all PatchCPGs of a patch to PatchGNN. However, the cross-function relations are not well captured, and we plan to address this issue in our future work.

VIII. Discussions

Usability. Due to different maintenance habits and policies, the security-related commits in OSS may not be well documented. When the commit messages are not accurate or not available, GraphSPD outperforms TwinRNN by capturing richer semantic information from the source code. GraphSPD can be integrated into version control platforms like GitHub for guiding contributors to explicitly label their security commits and speed up pushing to different branches and/or versions. Note that our system is platform-independent, since we only utilize source code without any requirements on patch format. Our system can also assist users to decide the importance and urgency of single-purpose commits/patches, which can reduce the time window of being exploited. In addition, our system is feasible for developers who adopt third-party libraries to identify security patches in the upstream dependencies. GraphSPD can timely alert developers to make appropriate security fixes, even if the downstream software is highly customized.

Limitations and Future Work. Although GraphSPD only supports patches in C/C++ that has the most vulnerabilities [31], the architecture can be adapted to other programming languages by extending code analyzer to support corresponding syntax and structures. However, it may be challenging to collect enough samples for safe languages like Rust.

Our method is also limited to the training datasets that are unable to cover all security patches in the real world. Specifically, PatchDB only collects security patches from NVD and

specific GitHub repositories, and SPI-DB contains security patches only from two repositories, though they are the largest-size publicly available datasets, to the best of our knowledge. As a learning-based method, GraphSPD mainly learns from the existing patterns and may not apply to the unseen ones, and thus their performance is affected by the distributions of the training data. In contrast, rule-based methods are driven by the inherent rules and are not affected by the datasets, but they may not work on samples out of the existing rules. Our future work will improve model generalization ability and use GraphSPD to help collect more security patches.

Our current method cannot distinguish specific security patch types. The root cause lies in two facts: (1) some patches are too rare to be included in training set; (2) data is imbalanced in security patch datasets. According to the analysis on NVD [39], 24.6% of vulnerabilities are related to code execution, whereas only 0.1% of vulnerabilities are HTTP response splitting. To mitigate data imbalance, we can adopt undersampling (e.g., Tomek Links [40]) or oversampling (e.g., SMOTE [41]) methods. Another research direction is to train a model for each patch type since our model can be directly transferred to specific patch detection. However, this scheme leaves out the general patterns over security patches and may be subject to limited data of specific patches. Also, if fed with more data, GraphSPD can be further fine-tuned to achieve better performance in identifying patch types.

Moreover, due to the restrictions of adopted static analysis tool, we do not consider the call relations among multiple functions or the semantics of callee functions. Thus, we cannot well handle the patches that refactor statements among multiple functions or only change the function calls. Also, it cannot process some macros and inline assembly. As future work, we will extend our model to apply to more situations.

IX. Related Work

Patch Analysis. Li et al. [42] conducted an empirical study of security patches, revealing multiple significant behaviors. Soto et al. [43] conducted a large-scale study on Java patches, providing insights into high-quality automated code repair. To counter vulnerability-contributing commits, VCCFinder [44] was implemented to flag suspicious patches using SVM. Tian et al. [6] utilized textual and code features to detect bug fixing patches in Linux. SPIDER [45] identified safe patches that cannot interfere with normal program flows on valid inputs. Wu et al. and Huang et al. [46], [47] developed rule-based methods to identify some common types of security patches. Vulmet [48] can automatically generate hot patches for Android via learning patch semantics. Wang et al. [7] further used random forest with extracted patch features to classify security patches into specific vulnerability types. Recent studies have also taken advantage of deep learning for patch analysis [49], [50], [51]. PatchRNN [8] and SPI [9] identify security patches with RNN models. Researchers also conduct binary patch analysis, including binary diffing [52], [53], [54], patch presence test [55], [56], [57], [58], patch identification [59] and automated binary patching [60], [61].

Vulnerability Detection. To detect software vulnerabilities, recent studies include fuzzing based methods [62], [63], [64], behavior based methods [65], clone based methods [23], [37], and deep learning based methods [24], [35], [66], [67], [68], [69]. By fuzzing, IoTFuzzer [62] can find memory corruption while ACHyb [63] can detect kernel access control vulnerabilities. SemFuzz recovers vulnerability-related knowledge and generates PoC exploits [64]. However, fuzzing involves vast resource and time consumption and cannot cover all possible cases [70]. Digtool detects kernel vulnerabilities by monitoring dynamic behaviors of kernel execution [65]. VUDDY detects vulnerabilities based on code clones [23]. MVP detects vulnerabilities by signature matching [37]. Russell et al. [66] embedded code tokens and leveraged ML models to detect vulnerabilities. VulDeePecker [24], [35] represents a program as code gadgets (i.e., statements mutually related in semantics) and implements a vulnerability detection system with RNN.

Code Analysis via Graph Learning. It is natural to represent code as graphs because of the structure of program flows [12], [13], [71]. Due to the structured property, graphs can be utilized for automatic source code summarization [72], learning semantic program embeddings [73], and learning compiler optimization tasks [74]. Also, code similarity can be measured by graph learning, e.g., graph matching network [17], graph-graph interactions [16], and graph-based twin network [75]. Researchers further leveraged the graph-based code similarity to detect the buggy code [18]. VGraph [29] detects buggy code reuse by matching a set of code property triplets with a database. Graph learning can also be utilized in vulnerability detection for different languages [76] and applications [77]. CPGVA [14] performs vulnerability detection based on code property graphs. Devign [19] detects vulnerabilities by using a richer set of code semantic representations, including natural code sequence. For binary applications, Bin2vec [78] learns a representation of binary programs via GNNs. Graph learning is also used in binary code similarity detection [79], [80]. Asteria [79] detects binary similarity via the AST semantics. Similarly, Zhu et al. [80] detected cross-platform binary similarity using neural machine translation with graph embedding.

X. Conclusion

This paper presents a graph-based detection system named GraphSPD to detect security patches with higher accuracy and fewer false alarms. GraphSPD is performed in two steps: transforming patches into PatchCPGs and detecting security patches via PatchGNN. A PatchCPG is constructed by merging the pre-patch and post-patch CPGs, reflecting the control/data dependencies and program syntax as well as the code version information. We build PatchGNN with a multi-attributed convolution mechanism to adapt the diverse attributes in PatchCPGs. The experimental results show that GraphSPD can achieve up to 80% accuracy when identifying security patches in various repositories, with a low false-positive rate of 5%. In addition, our case study on four real-word projects further validates the effectiveness and practicality of GraphSPD by identifying 88 silent security patches.

ACKNOWLEDGMENTS

This work was partially supported by the US Department of the Army grant W56KGU-20-C-0008, the US Office of Naval Research grants N00014-18-2893, N00014-18-1-2670, N00014-20-1-2407, and the US National Science Foundation grants CNS-1815650 and CNS-1822094, the Commonwealth Cyber Initiative, and NSFC grant 62132011.

REFERENCES

[1] Synopsys technology. 2021 Open Source Security and Risk Analysis Report. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html.

[2] National Institute of Standards and Technology. National Vulnerability Database. https://nvd.nist.gov/.

[3] National Vulnerability Database. CVE-2021-22205 Detail. https://nvd.nist.gov/vuln/detail/CVE-2021-22205.

[4] Kami Vaniea and Yasmeen Rashidi. Tales of software updates: The process of updating software. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, page 3215–3226, New York, NY, USA, 2016. Association for Computing Machinery.

[5] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. Detecting "0-day" vulnerability: An empirical study of secret security patch in oss. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 485–492, 2019.

[6] Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 386–396, 2012.

[7] Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2020.

[8] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. PatchRNN: A deep learning-based system for security patch identification. *CoRR*, abs/2108.03358, 2021.

[9] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. SPI: Automated identification of security patches via commits. *ACM Trans. Softw. Eng. Methodol.*, 31(1), September 2021.

[10] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. PatchDB: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 149–160, 2021.

[11] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[12] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[13] Wenhan Wang, Kechi Zhang, Ge Li, and Zhi Jin. Learning to represent programs with heterogeneous graphs. *CoRR*, abs/2012.04188, 2020.

[14] Wang Xiaomeng, Zhang Tao, Wu Runpu, Xin Wei, and Hou Changyu. CPGVA: Code property graph based vulnerability analysis by deep learning. In *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*, pages 184–188, 2018.

[15] Zhibin Guan, Xiaomeng Wang, Wei Xin, and Jiajie Wang. Code property graph-based vulnerability dataset generation for source code detection. In Guangquan Xu, Kaitai Liang, and Chunhua Su, editors, *Frontiers in Cyber Security*, pages 584–591, Singapore, 2020. Springer Singapore.

[16] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. SimGNN: A neural network approach to fast graph similarity computation. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, page 384–392, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 3835–3845. PMLR, 2019.

[18] Yuede Ji, Lei Cui, and H. Howie Huang. BugGraph: Differentiating source-binary code similarity with graph triplet-loss network. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 702–715, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[20] Cppcheck. https://cppcheck.sourceforge.io.

[21] David A. Wheeler). flawfinder. https://dwheeler.com/flawfinder/.

[22] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2012.

[23] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, 2017.

[24] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[25] David Golden. A survey of git best practices. https://xdg.me/a-survey-of-git-best-practices/.

[26] Perforce Software. 5 Best Git Practices for Git Commit. https://www.perforce.com/blog/vcs/git-best-practices-git-commit.

[27] NIST. NVD CWE Slice. https://nvd.nist.gov/vuln/categories.

[28] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[29] Benjamin Bowman and H. Howie Huang. VGRAPH: A robust vulnerable code clone detection system using code property triplets. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 53–69, 2020.

[30] The Joern Project. Code Property Graph — Joern Documentation. https://docs.joern.io/code-property-graph/, 2021. [Online; accessed 23-October-2021].

[31] White Source Software. The State of Open Source Vulnerabilities 2021. https://www.whitesourcesoftware.com/open-source-vulnerability-management-report/.

[32] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2014.

[33] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, SE-10(4):352–357, 1984.

[34] Wahab Khan, Ali Daud, Jamal Abdul Nasir, and Tehmina Amjad. A survey on the state-of-the-art machine learning models in the context of nlp. *kuwait journal of science*, 43, 2016.

[35] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawwei Zhu, and Zhaoxuan Chen. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2021.

[36] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[37] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182. USENIX Association, August 2020.

[38] The MITRE Corporation (MITRE) - Common Weakness Enumeration (CWE) . 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25, 2021.

[39] CVE Details (The ultimate security vulnerability datasource). Vulnerabilities By Type. https://www.cvedetails.com/vulnerabilities-by-types.php.

[40] Ivan Tomek. Two modifications of CNN. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(11):769–772, 1976.

[41] Alberto Fernández, Salvador García, Francisco Herrera, and Nitesh V. Chawla. SMOTE for learning from imbalanced data: progress and challenges, marking the 15-year anniversary. *J. Artif. Int. Res.*, 61(1):863–905, January 2018.

[42] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on*

Computer and Communications Security, CCS '17, page 2201–2215, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 512–515, New York, NY, USA, 2016. Association for Computing Machinery.

[44] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 426–437, New York, NY, USA, 2015. Association for Computing Machinery.

[45] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. SPIDER: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1562–1579, 2020.

[46] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[47] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 539–554. IEEE, 2019.

[48] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2397–2414. USENIX Association, August 2020.

[49] Thong Hoang, Julia Lawall, Richard J. Oentaryo, Yuan Tian, and David Lo. PatchNet: A tool for deep patch classification. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 83–86, 2019.

[50] Thong Hoang, Julia Lawall, Yuan Tian, Richard Jayadi Oentaryo, and David Lo. PatchNet: Hierarchical deep learning-based stable patch identification for the linux kernel. *CoRR*, abs/1911.03576, 2019.

[51] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 981–992, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 253–270, Vancouver, BC, August 2017. USENIX Association.

[53] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning program-wide code representations for binary diffing. *Network and Distributed System Security Symposium*, 2020.

[54] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. PatchScope: Memory object centric patch diffing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 149–165, New York, NY, USA, 2020. Association for Computing Machinery.

[55] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. BScout: Direct whole patch presence test for java executables. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1147–1164. USENIX Association, August 2020.

[56] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3649–3666. USENIX Association, August 2021.

[57] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902, Baltimore, MD, August 2018. USENIX Association.

[58] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. PDiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1149–1163, New York, NY, USA, 2020. Association for Computing Machinery.

[59] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. SPAIN: Security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472, 2017.

[60] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. Automating patching of vulnerable open-source software versions in application binaries. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[61] Christian Niesler, Sebastian Surminski, and Lucas Davi. HERA: Hotpatching of embedded real-time applications. *Network and Distributed System Security Symposium*, 2021.

[62] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[63] Yang Hu, Wenxi Wang, Casen Hunger, Riley Wood, Sarfraz Khurshid, and Mohit Tiwari. ACHyb: A hybrid analysis approach to detect kernel access control vulnerabilities. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 316–327, New York, USA, 2021. Association for Computing Machinery.

[64] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2139–2154, New York, USA, 2017. Association for Computing Machinery.

[65] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 149–165, Vancouver, BC, August 2017. USENIX Association.

[66] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762, 2018.

[67] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 914–919, New York, NY, USA, 2017. Association for Computing Machinery.

[68] Dipok Chandra Das and Md. Rayhanur Rahman. Security and performance bug reports identification with class-imbalance sampling and feature selection. In *2018 Joint 7th International Conference on Informatics, Electronics Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision Pattern Recognition (icIVPR)*, pages 316–321, 2018.

[69] Katerina Goseva-Popstojanova and Jacob Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 344–355, 2018.

[70] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[71] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2020.

[72] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 184–195, New York, NY, USA, 2020. Association for Computing Machinery.

[73] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. Learning semantic program embeddings with graph interval neural network. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[74] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on*

Compiler Construction, CC 2020, page 201–211, New York, NY, USA, 2020. Association for Computing Machinery.

[75] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marcus, Nesime Tatbul, Jesmin Jahan Tithi, Paul Petersen, Timothy G. Mattson, Tim Kraska, Pradeep Dubey, Vivek Sarkar, and Justin Gottschlich. MISIM: an end-to-end neural code similarity system. *CoRR*, abs/2006.05265, 2020.

[76] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2021.

[77] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge & Data Engineering*, (01):1–1, jul 5555.

[78] Shushan Arakelyan, Christophe Hauser, Erik Kline, and Aram Galstyan. Bin2vec: Learning representations of binary executable programs for security tasks. *CoRR*, abs/2002.03388, 2020.

[79] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection. *CoRR*, abs/2108.06082, 2021.

[80] Xiaodong Zhu, Liehui Jiang, and Zeng Chen. Cross-platform binary code similarity detection based on nmt and graph embedding. *Mathematical Biosciences and Engineering*, 18:4528–4551, 05 2021.

APPENDIX A
NETWORK ARCHITECTURE OF PATCHGNN

The detailed network architecture of PatchGNN is listed in Table VIII. Note that *MultiAttrConv* contains 5 *GraphConv* layers, which reduce the dimension by half.

TABLE VIII: PatchGNN network architecture in GraphSPD.

Layer	Input	Output	Comments
MultiAttrConv1	20	50	1/2, concat aggregate
MultiAttrConv2	50	25	1/2, mean aggregate
MultiAttrConv3	25	12	1/2, mean aggregate
Maxpool	12	12	
Meanpool	12	12	
Concat	12	24	max + mean pooling
Dropout	24	24	p = 0.5, only for training
FC1	24	8	bias = True
ReLU	8	8	
FC2	8	2	bias = True
Softmax	2	2	

The PatchGNN training curve is illustrated in Figure 7.

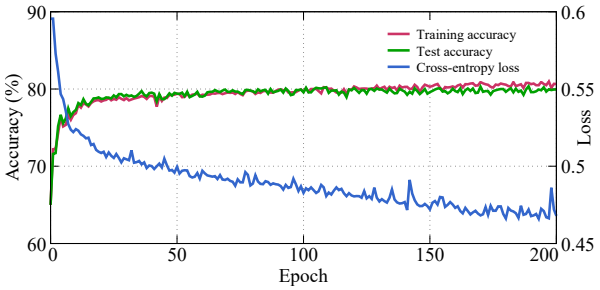
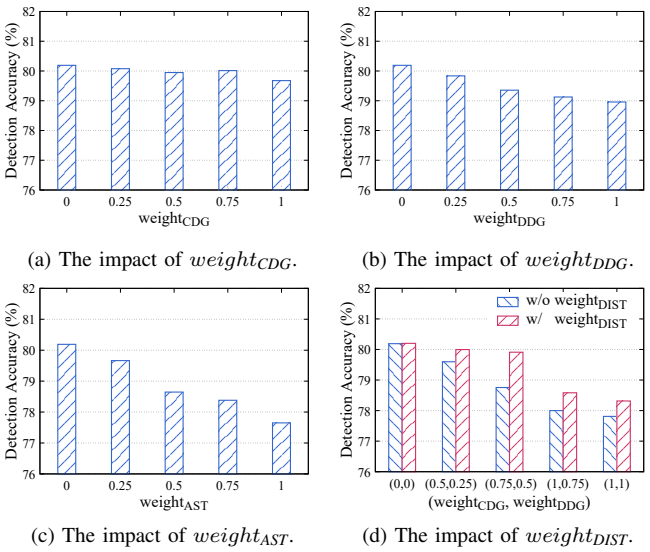


Fig. 7: The training curve of the PatchGNN model.

APPENDIX B
COMPLEMENTARY ANALYSIS ON CONTEXT STATEMENTS.

As described in Section IV, there is a trade-off between semantics and noise when considering context statements in PatchCPGs. Besides the program slicing scheme, we also try other ways to reduce the impact of irrelevant nodes and prevent over-fitting. One attempt is to use different weights to present the importance of context nodes. That is to say, for the PatchCPGs without slicing, we set weights (≤ 1) to the context nodes, artificially intervening the contextual information.



(a) The impact of $weight_{CDG}$. (b) The impact of $weight_{DDG}$.

(c) The impact of $weight_{AST}$. (d) The impact of $weight_{DIST}$.

Fig. 8: The impact of different weights for the context nodes.

1) Set weights based on different types of context nodes. Not all the nodes in a graph contribute equal to the predictions. Due to the different roles of CDG, DDG, and AST, a straightforward idea is to set 3 *component weights* (i.e., $weight_{CDG}$, $weight_{DDG}$, and $weight_{AST}$) for different context nodes. In the cases that a context node exists in two or more components, the weight is determined by the largest component weight applicable to this node. To evaluate the impact of different component weights, we utilize the control variates method: we measure the detection accuracy by setting one weight to different values and leaving others to 0s. The experimental results are shown in Figure 8(a)-(c). We conclude our observations as follows:

(a) The weighting scheme has lower accuracy than code slicing since we still retain all irrelevant nodes without any control/data dependency with the changed code.

(b) The detection accuracy will reduce with the increase of all these component weights.

(c) Compared with CDG, the accuracy can drop more significantly with higher DDG or AST weights.

(d) A higher $weight_{AST}$ will dramatically reduce the accuracy since too many AST nodes can lead to over-fitting (see the 4th finding in Appendix D). Thus, it is better to remove AST components for context nodes (i.e., $weight_{AST} = 0$).

2) Set weights based on the distance of context nodes. Another trial is to decide the weights according to the distance of context nodes towards the added/deleted nodes (i.e., hop count). The objective of this method is to assign larger weights for more direct context. We define a *distance weight* as $weight_{DIST} = 1/(1 + d)$, where d is the min hop count between a context node and changed nodes, e.g., the distance weight of 1-hop context nodes is 1/2. In this trial, the context node weight is the product of component weight and distance weight. We conduct multiple comparative experiments to an-

alyze the impact of $weight_{DIST}$ with different combinations of component weights. Based on the previous observations, we set $weight_{AST} = 0$ and $weight_{CDG} \geq weight_{DDG}$ in the experiments. In Figure 8(d), we find the distance weight can effectively boost the performance because it inhibits the effect of irrelative context nodes far away from the changed ones. However, since we achieve the best performance with 1-hop context only, we do not add context weights in our final design.

Appendix C
Pattern Analysis of Security Patches

We find some security patch types (e.g., resource leakage, NULL pointer dereference, race condition, and double free/use after free) exhibit distinguishable features. Accordingly, experiments show GraphSPD achieves a higher TP rate in these specific types. We further look into each patch type and summarize the unique patterns, which are reflected in our feature extraction of PatchCPG (Section V-A).

1) Resource Leakage. We find 3 patterns for resource leakage: re-initialization, memory operations, and file operations.

(a) **Re-initialization.** An effective method to mitigate memory leakage vulnerability is to re-initialize the memory space to default values, such as 0s. An example:
```
+ memset(&link, 0, sizeof(link));
```

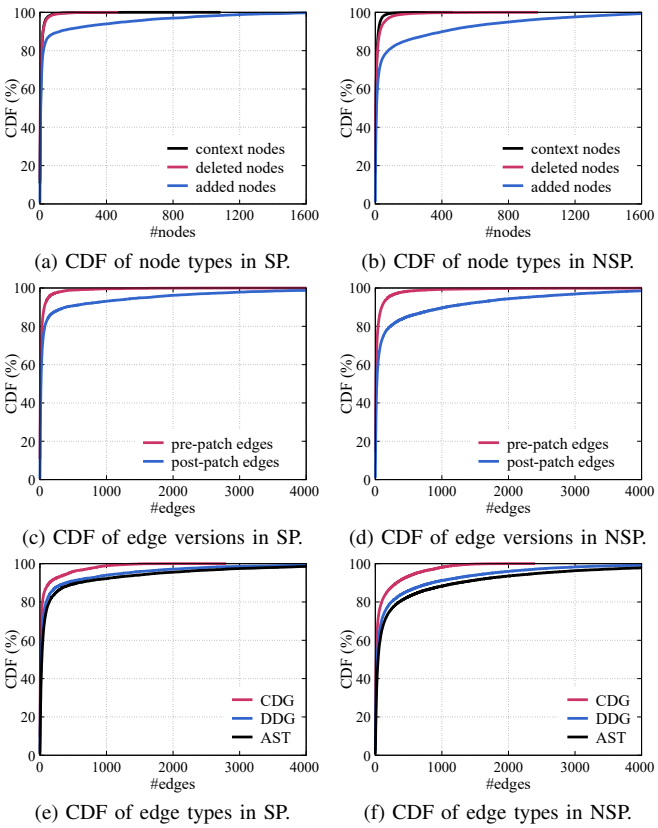
(b) **Memory Release.** Resource leakage may occur when software developers forget to release the memory. Thus, the corresponding patches will involve memory function calls such as `release()` and `free()`. An example:
```
+ free(*appliance);
```

(c) **File Close.** Information can be leaked via the unreleased file pointer. Therefore, the corresponding patches will involve file function calls such as `fclose()`. An example:
```
+ if (file != NULL)
+     fclose(file);
```

2) NULL Pointer Dereference. The vulnerability of NULL pointer dereference occurs when using an invalid pointer without checking its value. Therefore, the patches will use sanity checks (`if` statements) to verify the validity of pointer before usage. Moreover, the conditions always involve NULL, which accords with our feature of null identifiers. Example:
```
+  if (tmpjobid == NULL)
+      return(NULL);
```

3) Race Condition. Race condition occurs when the system attempts to execute two or more operations simultaneously. The effective method to avoid race condition is to use lock/unlock operations to restrict the processes and threads. Therefore, the patches are usually related to the lock APIs. An example:
```
+  mutex_lock(&pool->pool_lock);
   list_add(&ce->node, &pool->curlring);
   pool->curls++;
+  mutex_unlock(&pool->pool_lock);
```

4) Double Free/Use After Free. Double free occurs when `free()` is called more than once with the same memory address. Use-after-free occurs when using an invalid memory that has already been freed. We sum up 4 common patterns:



(a) CDF of node types in SP. (b) CDF of node types in NSP.
(c) CDF of edge versions in SP. (d) CDF of edge versions in NSP.
(e) CDF of edge types in SP. (f) CDF of edge types in NSP.

Fig. 9: The CDFs of nodes and edges in security patches (SP) and non-security patches (NSP).

(a) **Remove Second Use/Free.** For double free, a straightforward fixing method is to remove the extra free operation. The patch is related to the memory APIs. An example:
```
- free(comment_header);
```

(b) **Move Use Before Free.** For the use-after-free, some patches will move the statements to the lines before the free operation. We detect this type of security patches using the control dependency in PatchCPG. An example:
```
+  atomic_dec(&chip->active);
   if (!chip->num_interfaces)
       snd_card_free(chip->card);
-  atomic_dec(&chip->active);
```

(c) **Initialize Pointer to NULL.** To prevent using an invalid pointer, developers can initialize the pointer to NULL and use it. An example:
```
+ info->port = NULL;
```

(d) **Check Before Use.** To exclude invalid pointers, developers can check the validity before using the pointer. The patches can add corresponding sanity check. An example:
```
- *image = (rbd_image_t)ictx;
+ if (r >= 0)
+    *image = (rbd_image_t)ictx;
```

Appendix D
Statistical Properties about PatchCPG

We try to analyze the statistical properties about PatchCPGs from a broad perspective. Our goal is to find some patterns of patches by observing the statistical results. We measure the

cumulative distribution functions (CDFs) about 8 graph components for security and non-security patches, respectively. These graph components include 3 types of nodes (i.e., 1-hop context, added, deleted) and 5 types of edges (i.e., pre-patch, post-patch, CDG, DDG, AST). From the statistical results in Figure 9, some interesting findings have already emerged.

1) **80% of graphs are quite small.** In Figure 9, we find 80% of PatchCPGs have fewer than 100 nodes and fewer than 200 edges, which is crucial for the PatchGNN model to prevent over-fitting. More complex graphs mean more complex information processing, raising the requirement to increase model complexity and enlarge training dataset.
2) **Security patches have a smaller graph size.** Statistically, the mean node (edge) amount of security patches is smaller than that of non-security ones. The reason is that security patches focus on fixing vulnerabilities and only modify a few critical statements. Non-security patches usually add new functionality features, thus involving more statement changes. However, the size distributions of two patch types do not support a certain threshold to distinguish security and non-security patches.
3) **Significantly more nodes are added than are deleted.** For both security and non-security patches, more nodes and edges emerge in the post-patch graphs since developers tend to add new statements into the source code. That might be the reason why the software always become more and more bloated after the updates.
4) **AST is more numerous than DDG and CDG.** We find CDG and DDG are fewer in PatchCPG while AST is the most common type. We can even ignore the AST with limited resources because the AST information is already contained in the main statement nodes (the roots of AST) and high graph complexity can easily lead to over-fitting.

APPENDIX E
DETECTED SECURITY PATCHES IN NGINX, XEN, OPENSSL, AND IMAGEMAGICK

In Section VII, GraphSPD detects 21 security patches in NGINX. However, none of them have been assigned with a corresponding CVE ID or are documented as a security fix in the NGINX's changelog. Moreover, 6 out of 21 commits only have a commit subject without detailed commit messages, and 4 commits do not mention security impacts in the subjects or messages. Similarly, other security commits detected from the GitHub repositories of Xen, OpenSSL, and ImageMagick by the GraphSPD are also not linked with a CVE entry or explicitly documented with potential security impacts.

Note that we do not mean that maintainers of NGINX or others intend to hide their security patches. Instead, we understand that the patch labeling is subjective and maintainers may think CVEs are unnecessary for these bugs/vulnerabilities.

In practice, the CVE assignment is quite subjective and sometimes inconsistent among different CVE Numbering Authorities (CNAs). For instance, even for the same memory-cache side-channel vulnerability in both LibreSSL and OpenSSL, the CVE Numbering Authority of Last Resort (CNA-LR) created CVE-2018-12434 for LibreSSL, but OpenSSL's CNA refused to assign a CVE for it [5].

Besides CVE, maintainers can also use changelog and commit messages to recognize a security patch. But, due to different maintenance habits or policies, some security impacts may not be documented well (e.g., inaccurate terminology or jargon), which makes users hard to locate these security patches for prioritized use. Given the fact that maintainers of software repositories may not provide sufficient security-related information explicitly, our system can help admins be more aware of potential security-related patches.

In Table IX, we list all the security patches successfully detected by the GraphSPD with their commit IDs and manually analyze their corresponding vulnerability types.

TABLE IX: List of detected security patches in Xen, NGINX, OpenSSL, and ImageMagick.

Repo	Commit ID	Vuln Type	Commit ID	Vuln Type
Xen	d670ef3401	uninitialized use	1ef48c82e7	use after free
	29fae90baa	null pointer deref	939775cfd3	null pointer deref
	ff522e2e91	buffer overflow	9dc46386d8	uninitialized use
	e0ca7b883a	null pointer deref	5a3f7a05a3	null pointer deref
	f2c620aa06	uninitialized use	0bdaa8b035	null pointer deref
	8e76aef728	race condition	243036df0d	null pointer deref
	ff3e7e7681	null pointer deref	cd09c4929e	resource leak
	79b6574f8e	resource leak	8a62dee9ce	resource leak
NGINX	60a8ed26f3	resource leak	9a3ec20232	resource mgmt err
	327e21c432	infinite loop	cfa669151e	resource mgmt err
	bd7dad5b0e	buffer overflow	4ee66b3f7b	resource leak
	4cd1dd28dd	use after free	c3fd5f7e76	resource leak
	36dfa020f2	resource leak	5784889fb9	null pointer deref
	52d9da8790	resource leak	7e3041b79f	use after free
	278be041dd	infinite loop	dac90a4bff	resource leak
	4ac8036e78	resource leak	cdbdbbd842	resource mgmt err
	661e40864f	resource mgmt err	92111c92e5	buffer overflow
	b0b24e8a30	resource leak	aa04b091ae	null pointer deref
	9359155b2f	uninitialized use		
OpenSSL	1832bb0f02	integer overflow	885d97fbf8	uninitialized use
	b2f90e93a0	resource leak	fb0f65fff8	race condition
	b3c34401c0	resource leak	f99b34957f	double free
	7a85dd46e0	null pointer deref	1b4d9967a2	resource leak
	10481d3384	resource leak	dc7e42c6a1	null pointer deref
	68b78dd7e4	null pointer deref	79cda38cff	resource leak
	04e3ab64d5	null pointer deref	5266af8737	null pointer deref
	c81eed84e4	resource leak	8d215738a0	null pointer deref
	09dca55733	null pointer deref	0ce0c45586	resource leak
	da7db7ae6d	resource leak	0c5905581e	null pointer deref
	7f1cb465c1	null pointer deref	ab547fc005	null pointer deref
	922422119d	resource leak	814999cb44	race condition
	74b485848a	resource leak	963eb12dbd	resource leak
	8c590a219f	null pointer deref	5327da81f0	resource leak
	fa17f5c987	resource leak	981a5b7ce3	resource leak
	f5c0f69619	null pointer deref	e20fc2ee4f	buffer overflow
	0449702abc	double free	70cd9a5191	uninitialized use
	b9648f31a4	null pointer deref	8086b267fb	resource leak
	6889ebff01	resource leak	433e13455e	null pointer deref
	366a162639	resource leak	7ca3bf792a	double free
	2823e2e1d3	buffer overflow	4e92d5c79d	buffer overflow
	09f38299cc	resource leak	ed5b26ce0b	null pointer deref
	09134f183f	null pointer deref		
Image-Magick	d61dd34fe0	resource mgmt err	3dc9db61ac	resource leak
	8a41ce827d	uninitialized use	f9c35c91ba	resource leak
	2df3d0124b	resource leak	225b51d7f2	resource leak