BinGo: Identifying Security Patches in Binary Code with Graph Representation Learning

Xu He
George Mason University
Fairfax, Virginia, USA
xhe6@gmu.edu

Shu Wang
George Mason University
Fairfax, Virginia, USA
swang47@gmu.edu

Pengbin Feng*
Xidian University
Xi'an, Shanxi, China
pbfeng@xidian.edu.cn

Xinda Wang
The University of Texas at Dallas
Richardson, Texas, USA
xinda.wang@utdallas.edu

Shiyu Sun
George Mason University
Fairfax, Virginia, USA
ssun20@gmu.edu

Qi Li
Tsinghua University
Beijing, China
qli01@tsinghua.edu.cn

Kun Sun
George Mason University
Fairfax, Virginia, USA
ksun3@gmu.edu

ABSTRACT

A timely software update is vital to combat the increasing security vulnerabilities. However, some software vendors may secretly patch their vulnerabilities without creating CVE entries or even describing the security issue in their change log. Thus, it is critical to identify these hidden security patches and defeat potential N-day attacks. Researchers have employed various machine learning techniques to identify security patches in open-source software, leveraging the syntax and semantic features of the software changes and commit messages. However, all these solutions cannot be directly applied to the binary code, whose instructions and program flow may dramatically vary due to different compilation configurations. In this paper, we propose BinGo, a new security patch detection system for binary code. The main idea is to present the binary code as code property graphs to enable a comprehensive understanding of program flow and perform a language model over each basic block of binary code to catch the instruction semantics. BinGo consists of four phases, namely, patch data pre-processing, graph extraction, embedding generation, and graph representation learning. Due to the lack of an existing binary security patch dataset, we construct such a dataset by compiling the pre-patch and post-patch source code of the Linux kernel. Our experimental results show BinGo can achieve up to 80.77% accuracy in identifying security patches between two neighboring versions of binary code. Moreover, BinGo can effectively reduce the false positives and false negatives caused by the different compilers and optimization levels.

*Pengbin Feng is the corresponding author.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

Security Patch, Binary Program, Language Model, Graph Learning

ACM Reference Format:
Xu He, Shu Wang, Pengbin Feng, Xinda Wang, Shiyu Sun, Qi Li, and Kun Sun. 2024. BinGo: Identifying Security Patches in Binary Code with Graph Representation Learning. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3634737.3637666

1 INTRODUCTION

Software patching is a common practice in software maintenance to ensure the stability, performance, and security of software. Thus, software developers periodically release patch packages to add new features, address performance bugs, and fix security vulnerabilities. Among them, security patches should be prioritized to address software vulnerabilities and thus prevent N-day threats coming from adversaries. However, timely deployment of security patches remains a challenge since not all security patches are explicitly marked [56]. For example, software vendors may secretly patch their vulnerabilities without creating CVE entries or even describing the security issue in its change log [45]. Therefore, identifying these hidden security patches becomes critical for developers and users to improve their software security.

Researchers have leveraged various syntax and semantic features to help identify security patches, such as the software changes as well as their metadata [13, 26, 39, 43, 47–49, 51, 57]. However, those solutions require the availability of source code commits that include source code diffs and commit messages, and unfortunately, not all patches are explicitly released with accessible source code. Particularly, commercial closed-source software vendors usually only release a new version of the binary file to replace the old version, presenting challenges in demystifying patch details.

The existing security patch detection methods on source code cannot be directly applied to the binary code due to the limitation of representing the binary code as either a sequential model or a graph model. The sequential-based solution [51] treats the changed traces (i.e., a sequence of basic blocks or instructions) as the patch pattern. Alternatively, graph-based solutions try to catch the control flow and data dependency by representing source code as code property graphs [6, 18, 25, 43, 49]. However, the analysis of binary code is more complex than that of source code, since both the instruction sequence and the program flow in binary code may dramatically vary due to different compilation configurations (e.g., compiler versions and optimization levels) used in compiling the source code [16]. The sequential-based solutions can hardly accommodate the program flow changes due to the branch and loop statements (which are usually associated with software vulnerabilities [48]) in identifying binary security patches. Though recent studies [6, 43, 49] incorporate graph models to represent program flow, since only statistical features of statements are embedded in each node, they miss subtle code changes in security patches and trigger more false negatives.

In this paper, we propose BinGo, an end-to-end security patch detection system over binary code. The main idea is to present the code as code property graphs to enable a comprehensive understanding of program flow and perform a Language Model (LM)-based model over each basic block to catch the instruction semantics. BinGo consists of four phases, namely, patch data pre-processing, graph extraction, embedding generation, and graph representation learning.

BinGo first identifies and extracts patch-related code segments from a pair of pre-patch and post-patch binaries. Different from the source code analysis that regards code statements as process units, our binary code analysis treats basic blocks of assembly code as process units, which consist of a sequence of contiguous straight-line instructions without branches. Basic blocks are capable of better preserving the inherent code logic than individual assembly instructions, which only perform low-level simple operations, i.e., moving registers. Because we focus on analyzing the code changes between pre-patch and post-patch binaries, it is critical to identify the patch-related basic blocks to reduce the analysis scope. We modify the workflow of DeepBinDiff [12] to detect patch-related basic blocks with a low false positive rate.

Next, we transform the assembly code into a graph representation to contain rich syntax and semantic information. We use the code property graph (CPG) that accommodates the control flow graph (CFG), control dependency graph (CDG), and data dependency graph (DDG) in a unified graph. To reduce the graph size, we implement a graph slicing method to selectively remove context basic blocks that are too distant from the patch-related basic blocks in the control relations. In this step, we generate two code property graphs for pre-patch and post-patch binaries, respectively. The graph topology can be represented as an adjacent matrix, while the edge attributes can be embedded according to three relationships, i.e., CFG, CDG, and DDG. To fully embed the syntax and semantics in each node (i.e., basic block), we utilize an LM-based model to directly learn an embedding from the assembly instructions. Based on the language models, e.g., BERT [10], GPT [36], and BART [22], we can capture the real semantic, even if instructions in the node have been changed by compilation [14, 24, 26].

Finally, we develop a graph learning model to identify security patches. It is based on the siamese network architecture [29], which uses the same weights over both pre-patch and post-patch graphs to obtain a comparable output. Each branch of the model comprises three graph convolution layers to achieve a detailed understanding of the input graph representation. Since the graphs contain three different types of edges for CFG, CDG, and DDG, we propose a multi-head attention convolution mechanism that views each edge type as an individual convolution channel and aggregates the information of all channels to the next layer.

We implement a prototype of BinGo with 2,247 LoC in Python. Due to the absence of a binary security patch dataset, we first construct a benchmark by compiling the pre-patch and post-patch source code in the Linux kernel. This dataset can be further used in vulnerability detection, patch presence, and hot patch generation. Our experimental results show BinGo can achieve up to 80.8% accuracy with an F1 score of 0.76. Besides, BinGo can achieve a false negative ratio of 29.15% and a false positive ratio of 11.82%. We further demonstrate BinGo is effective under different compilation configurations, i.e., compilers and optimization levels, suggesting BinGo is a viable approach to alleviate the potential false positives caused by compilation factors.

In summary, we make the following contributions:

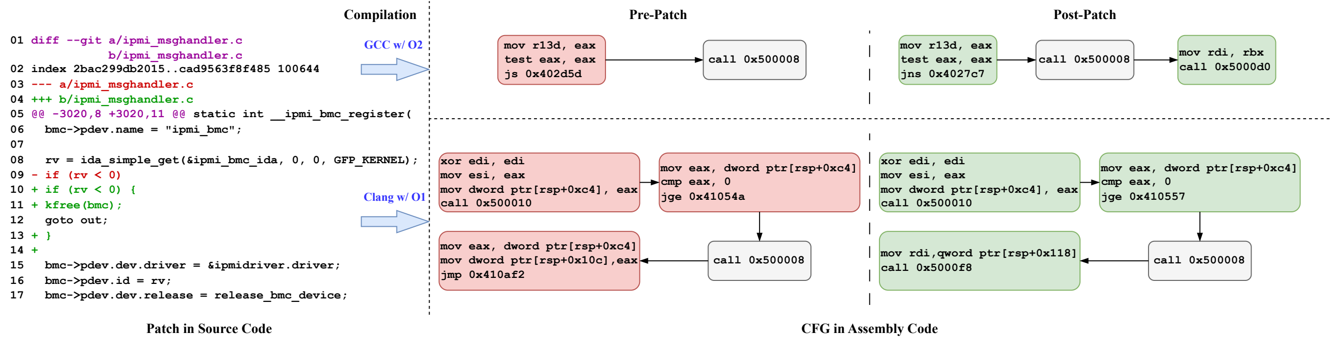
- We develop a new binary security patch detection system named BinGo, which can help users identify potential hidden security patches in newly released binary code and thus prioritize the related system update.
- We propose a new graph representation for binary code to integrate both the CPG-based static analysis features and the LM-based embedding features, providing a comprehensive representation of subtle code changes in binary code.
- We develop a graph learning-based detection model that adopts a siamese network to identify security patches by comparing the pre-patch and post-patch binaries.
- We implement a prototype of BinGo and evaluate its performance on our benchmark dataset. The experimental results show BinGo can achieve high accuracy as well as low false positives/negatives on identifying security patches between two neighboring versions of binary code.

2 PRELIMINARY

2.1 Problem Statement

Given two binary versions, we denote the pre-patch version as bin_a and the post-patch version as bin_b. Compared to bin_a, some basic blocks in bin_b have been modified, e.g., adding, deleting, or editing instructions, which can be used to fix vulnerabilities, address performance bugs, or add new functionalities. We categorize all code changes into two patch types, either the security patch denoted as p_s or the non-security patch denoted as p_{ns}. In this paper, our task is to develop a classification system f_c to check whether a given patch p is a security patch or a non-security one, as described in Formula (1).

$$f_c(p_i) = \{p_s | p_{ns}\}, w.r.t\{p_0, p_1, \ldots, p_n\} \in diff(bin_a, bin_b) \quad (1)$$



```
diff --git a/ipmi_msghandler.c
           b/ipmi_msghandler.c
index 2bac299db2015..cad9563f8f485 100644
--- a/ipmi_msghandler.c
+++ b/ipmi_msghandler.c
@@ -3020,8 +3020,11 @@ static int __ipmi_bmc_register(
  bmc->pdev.name = "ipmi_bmc";

  rv = ida_simple_get(&ipmi_bmc_ida, 0, 0, GFP_KERNEL);
- if (rv < 0)
+ if (rv < 0) {
+ kfree(bmc);
  goto out;
+ }
+
  bmc->pdev.dev.driver = &ipmidriver.driver;
  bmc->pdev.id = rv;
  bmc->pdev.dev.release = release_bmc_device;
```

```
mov r13d, eax
test eax, eax
js 0x402d5d
```

```
call 0x500008
```

```
mov r13d, eax
test eax, eax
jns 0x4027c7
```

```
call 0x500008
```

```
mov rdi, rbx
call 0x5000d0
```

```
xor edi, edi
mov esi, eax
mov dword ptr[rsp+0xc4], eax
call 0x500010
```

```
mov eax, dword ptr[rsp+0xc4]
cmp eax, 0
jge 0x41054a
```

```
call 0x500008
```

```
mov eax, dword ptr[rsp+0xc4]
mov dword ptr[rsp+0x10c],eax
jmp 0x410af2
```

```
xor edi, edi
mov esi, eax
mov dword ptr[rsp+0xc4], eax
call 0x500010
```

```
mov eax, dword ptr[rsp+0xc4]
cmp eax, 0
jge 0x410557
```

```
call 0x500008
```

```
mov rdi,qword ptr[rsp+0x118]
call 0x5000f8
```

Figure 1: A patch in source code and its corresponding assembly code with various compilation configurations (CVE-2019-19046).

To simplify the security patch identification process, we have two assumptions: (1) There is only one patch between bin_a and bin_b, which indicates that all modified basic blocks in $diff(bin_a, bin_b)$ are from the same patch; (2) A security patch only focuses on fixing one vulnerability, which implies that all patch-related basic blocks are connected together via data dependency or control dependency to resolve the vulnerability. We performed a preliminary study on 1,136 patches from the Linux kernel, among which 801 patches are security fixes. There are 661 patches among them only modified one file, and 469 patches only changed one function. Therefore, we believe that our assumption can cover most scenarios in practice.

2.2 Challenges of Security Patch Detection in Binary Code

Most open-source programs are maintained using Git in practice, which records the code changes before and after the patches [23], as shown in the source code part of Figure 1. Existing work [48] has shown that patches of different purposes can be distinguished based on the corresponding changes in control flow and data flow. For example, Wang *et al.* found that about 70% of security patches consist of if-then-else structures [46], which aim to add or modify the if conditions for security check addition or branch statement modification. However, for non-security patches, the common practice is to add or replace entire procedures (or even functions) to remove redundant code or add new features. In addition, the code size involved in security patches is usually much smaller than that of non-security patches. All these uncovered characteristics are very helpful to identify security patches and non-security patches at the source code level. Existing works [43, 57] have achieved good recognition performance by conducting multiple graph features and using graph learning algorithms.

However, for binary code, the compiler may introduce extra noise that increases the difficulty of security patch detection. Due to diverse compiler implementations (e.g., GCC [15] and Clang [8]) and optimization mechanisms (e.g., O0, O1, O2, O3, and Os), different binary code variants may be compiled from the same source code, although they execute exactly identical semantics [4, 16]. As shown in Figure 1, we take the patch of CVE-2019-10496 as an example to demonstrate such a difference. We compiled the Linux kernel for pre-patch and post-patch versions with two compilation configurations (i.e., Clang with O1 and GCC with O2). We retain the sliced CFG in assembly code, only including the basic blocks of code modification (i.e., the red and green ones) and necessary context. The source code shows that the patch merely adds a memory-releasing statement (i.e., `kfree(bmc)`) in the security check, but such a code change can involve multiple basic blocks in the assembly code. Furthermore, we observe that the CFGs derived from Clang and GCC have different instructions, basic blocks, and control dependencies.

With different compilation configurations, the mapping from source code to binary code would be different. Compiled by GCC with O2, the source code statement "`+ kfree(bmc)`" can result in a new basic block and a changed CFG structure in the binary code. However, compiled by Clang with O1, the changes only focus on the instructions and there is no effect on basic blocks and CFG structure. Therefore, when detecting security patches in binary code, we also need to consider the additional issues caused by compilation configurations. In this paper, BinGo constructs new semantic patterns by fusing assembly instruction embeddings and CPG-based graph representation to incorporate the changes on both code instructions and program flow in the binary patches. Moreover, we also build a binary patch dataset based on multiple compilation configurations to evaluate the effectiveness of the BinGo and its robustness to the compilation.

3 SYSTEM DESIGN

BinGo is an end-to-end deep learning model that detects security patches from non-security ones in binary code. Figure 2 shows the overview of BinGo, which consists of four phases: patch data pre-processing, graph extraction, embedding generation, and graph representation learning. First, BinGo extracts patch-related code snippets from pre-patch and post-patch binaries. Second, BinGo conducts the graph representation by connecting patch-related blocks according to their control flow and data flow dependencies. Third, BinGo converts attributes of nodes and edges in the graph into the numeric embedding representations. Finally, we feed the embeddings into the graph learning model to classify the security patches and the non-security patches.

3.1 Patch Data Pre-Processing

Given a pair of pre-patch and post-patch binary programs, the first phase is to extract the patch-related code segments. Considering the complex semantic mapping from source code to assembly code

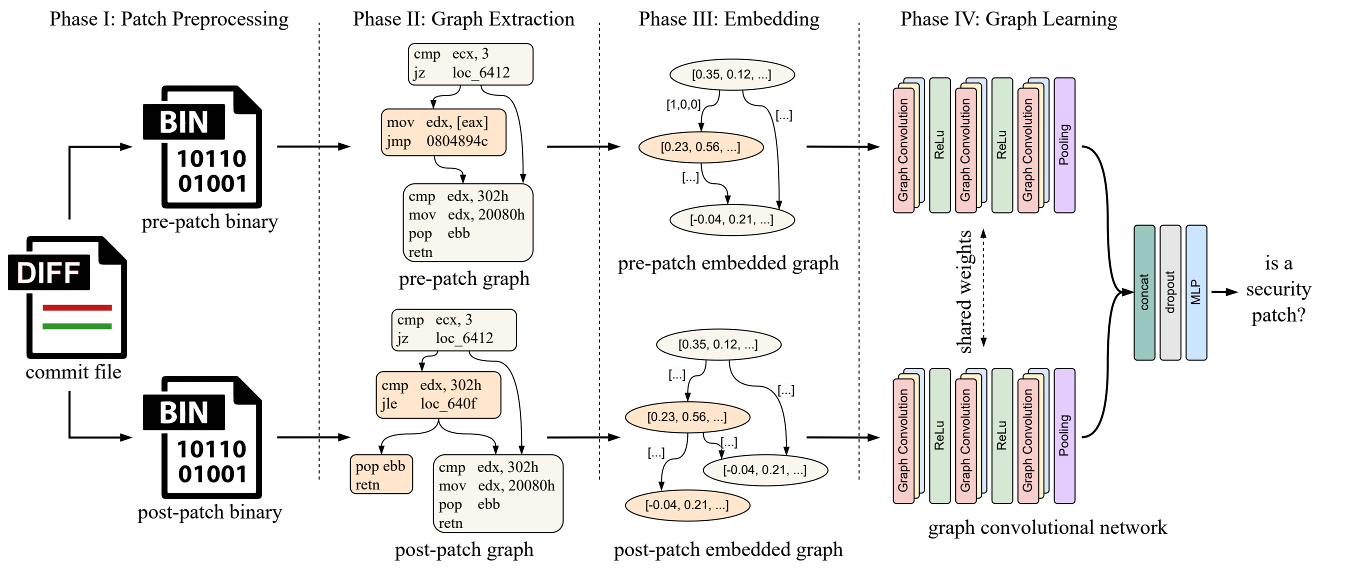


Figure 2: The overview of the BinGo system that contains four phases.

during the compilation, modifying one statement in the source code may correspond to one or even multiple basic blocks in the assembly code. For instance, the statement "`c = (a < b) ? a : b;`" can be split into 3 basic blocks. Thus, a binary patch can be represented by a set of basic blocks, rather than a single instruction or function.

Formally, we can define the basic block set in pre-patch binary as $BB_{pre} = \{bb_{a_0}, bb_{a_1}, \ldots, bb_{a_n}\}$, where bb_{a_i} represents a basic block containing instructions related to the patch. To compare the differences before and after the patch, we also define $BB_{post} = \{bb_{b_0}, bb_{b_1}, \ldots, bb_{b_m}\}$, which represents all corresponding basic blocks in post-patch binary.

A patch-related block refers to the basic block in which there is at least one changed (i.e., add, delete, or modify) instruction. Intuitively, we can detect such blocks by diffing the pre-patch and post-patch disassembled programs. In practice, we unexpectedly found some rule-based diff tools (e.g., BinDiff in IDA) have high false positives in detecting patch-related blocks [12]. We infer there are two reasons for this issue: (1) Basic block changes caused by compilation: the recompilation of modified source code will not only affect the direct-related blocks but also change the neighborhood blocks, especially compiled with a higher optimization level. (2) The basic blocks of pre-patch and post-patch programs are not in one-to-one correspondence: the modification of a source code statement may correspond to one or even multiple basic blocks.

For arbitrary binaries. With the comparison between the existing diff tools, we choose DeepBinDiff [12] to extract the patch-related blocks. The original DeepBinDiff directly detects basic blocks with distinct semantics by matching inter-procedural control-flow graphs (ICFG) in the program-wide code representation, which is precise but leads to high overhead. Thus, we modify the workflow of DeepBinDiff to reduce the false positive rate and improve computation efficiency. As shown in Figure 4(a), our modified DeepBindiff first narrows down the search scope to the function level and then filters discrepant basic blocks within patch-related functions. Such an approach works for arbitrary binaries, even for striped binaries, which do not include symbol table information.

For images with the symbol table. Our BinGo can also be used to identify security patches within released images. For pre-patch and post-patch images, the workflow of patch-related block extraction is shown in Figure 4(b). As the symbol table is included in most Linux-based kernel images [54], we can easily locate the functions by checking the entities with the same name. Then, we perform syntax-based function similarity detection method to filter the identical functions. Finally, we utilize the modified DeepBinDiff to extract patch-related blocks within functions.

For the ground truth in the Benchmark. We leverage the information of source code changes in commit messages and the debug information within binaries to locate patch-related blocks precisely. Specifically, we build a mapping between instruction and source code line number by using the DWARF information [38]. Then, we could precisely locate each patch-related block according to the source code changes.

3.2 Graph Extraction

In addition to patch-related block sets, the impact of patches on program semantics is also reflected in the changes in program flow and dependency. In other words, a patch also contains the relationship information between basic blocks. Code property graph (CPG) is a language-agnostic intermediate program representation, which merges multiple code graph representations into one queryable graph database [53]. To include all possible relations, our CPG-based representation merges three types of graphs (i.e., control flow graph (CFG), control dependence graph (CDG), and data dependency graph (DDG)) into a single joint data structure. As shown in Figure 3, CFG represents all the possible traversed paths

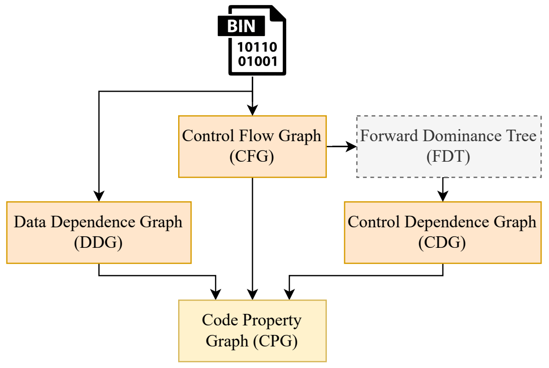

BIN
10110
01001
Control Flow Graph (CFG)
Forward Dominance Tree (FDT)
Data Dependence Graph (DDG)
Control Dependence Graph (CDG)
Code Property Graph (CPG)

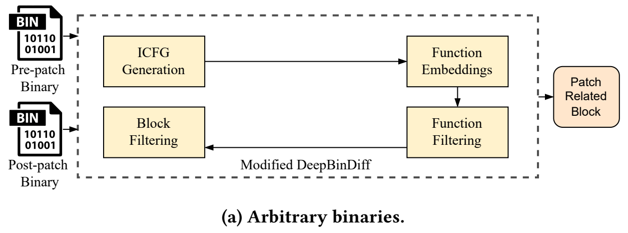
Figure 3: The code property graph (CPG) generation from the CFG, CDG, and DDG.

during program execution, and DDG represents the data access relation between statements. Noting that CDG is derived from CFG via inferring the forward dominance tree (FDT), which means that node B dominates node A if node A determines whether node B is executed. In our paper, CFG contains the direct dominance relationship, while CDG focuses on the indirect dominance relationships to avoid overlapping connections.

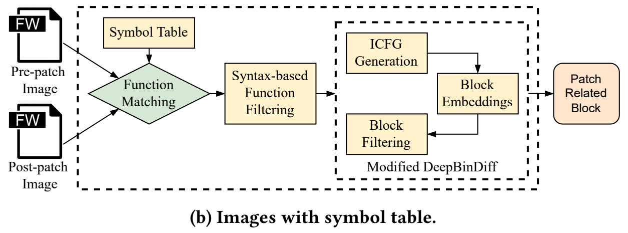
Given a patch-related block set, BinGo constructs the CPG-based graph in two steps. First, we derive a CFG to only connect all patch-related blocks, which indicates the internal connectivity of the patch. Second, we extend a larger graph to cover all patch-related blocks and connect them to their neighborhood basic blocks. That is, we add more context basic blocks to the graph, as long as these blocks are connected to patch-related blocks in the CFG, DDG, or CDG. Such a graph representation further implies the external impact of the patch. As shown in Phase II of Figure 2, the orange node in the graph denotes the patch-related blocks and the pale yellow node denotes the context blocks.

Graph Slicing. We adopt the graph slicing method to extract the relevant node connections and detect all relevant basic blocks in the CPG representation. Given a pair of pre-patch and post-patch binaries, we first extract their CFG, CDG and DDG from the assembly code. For the internal graph, we match the patch-related blocks in the program-wide CFG to retain internal connections. For the entire graph, we design a graph slicing algorithm.

The algorithm takes the patch-related block set (BB), the graph relations including CFG, CDG, and DDG ($\{G_{cfg}, G_{cdg}, G_{ddg}\}$), and the slice stride (n) as inputs, and outputs the sliced CPG-based representation. Traversing each basic block from the block set, BinGo first traces the control and data dependency relations between blocks. That is to find the predecessor and successor of the current block in CFG, CDG, and DDG. Correspondingly, We will create two collections ($forSlices$ and $backSlices$) to store the predecessors and successors of all blocks, respectively. Next, BinGo iteratively performs both forward and backward slicing to extract all neighborhood blocks. The slice stride n is a configurable parameter that represents the maximum distance from a context node to the nearest patch-related node. The slicing process will terminate after n iterations or until the sliced graph structure converges. Finally, we connect the patch-related blocks BB, the forward sliced blocks $forSlices$, and the backward sliced blocks $backSlices$ as the final CPG-based representation $\boldsymbol{G}$.



(a) Arbitrary binaries.



(b) Images with symbol table.

Figure 4: The extraction of the patch-related blocks from arbitrary binaries and the images with symbol table.

3.3 Embedding Generation

In the graph representation, the extracted nodes and edges cannot be directly fed into the deep learning model. Thus, we convert the graph topology into an adjacency matrix, which can represent the node connection information. For the edges and nodes, we embed their attributes into the numeric matrices, which are represented as node embedding $\boldsymbol{N}$ and edge embedding $\boldsymbol{E}$. Given a CPG representation $\boldsymbol{G} = \{\boldsymbol{E}, \boldsymbol{N}\}$, we first define an adjacency matrix $\boldsymbol{A}$ to denote the connectivity of $\boldsymbol{G}$, where $\boldsymbol{A}_{ij} = 1$ represents that node i and node j is connected, else $\boldsymbol{A}_{ij} = 0$ if they are unconnected. Because there are three subgraphs in the CPG, i.e., CFG, CDG, and DDG, $\boldsymbol{A}^{(k)}$ denotes the connectivity in the k-th subgraph. Note that, each subgraph in BinGo is a directed graph. For the edge representation, we denote the matrix of edge embeddings as $\boldsymbol{E}$, where $\boldsymbol{E}_d^{(k)}$ is the d-th edge in the k-th graph. $\boldsymbol{E}^{(k)}$ is a vector that contains all edges of the k-th subgraph. For the node representation, we denote the matrix of node embeddings as $\boldsymbol{N}$, where $\boldsymbol{N}_i$ is the i-th basic block in the CPG. Also, nodes can be shared by different subgraphs. For each node, we represent them with semantic embedding vectors.

A. Node Embedding

As shown in Phase III of Figure 2, we convert all nodes into the numeric representation, including the patch-related blocks and the context blocks. In binary code analysis, the most common numeric representation is the statistic-based features, such as counting distinct tokens in basic blocks. These tokens refer to operators and operands in instructions [2]. However, such an embedding method focuses more on the syntactic features but neglects the semantic information, which may be not fine enough for the subtle changes in security patches. Unlike source code, binary code is also subject to compiler optimizations. The same source code may be compiled into different binaries with different optimization configurations.

Therefore, the LM-based embedding model is utilized in BinGo due to its powerful semantic understanding capability [10, 14, 16, 22, 36].

Embedding Model. The BinGo employs a BERT-based embedding model [35] to learn the instruction semantics in the basic blocks. The BERT model can learn the context semantics of the instructions through specially designed self-supervised training tasks [24, 34].

Exploiting the transformer framework and attention mechanism, BERT can learn long-range contextual semantics of the instruction sequences in basic blocks. From the perspective of model structure, BERT is stacked by multiple transformer encoder units that share the same architecture. The BERT model consists of 12 transformer layers, which share the same architecture. As shown in Figure 5, the BERT tokenizes the input instruction sequences and encodes them as initial embeddings. The model will continue updating the embeddings via the transformer encoders. To distill the learning capability of the model on the instruction semantics, we need to train the model via well-designed training tasks. We denote the j-th layer embedding as EM_j and the encoder unit as f_e, then the output embedding in each encoder layer is expressed as $EM_{j+1} = f_e(EM_j)$.

Tokenization. Before we feed the instruction sequences into the embedding model, we need to tokenize them. The basic way is to decompose an instruction into one opcode and multiple operands. However, the operand can be further subdivided. For example, given an instruction "`mov rax, qword [rsp+0x58]`", we divide it into "`mov`", "`rax`", "`qword [rsp+0x58]`" originally. Note that the operand "`qword [rsp+0x58]`" is too complicated to cause an OOV (Out-Of-Vocabulary) problem. Thus, we subdivide operands into more basic tokens including registers (e.g., `rax` and `rsp`), constant (e.g., `0x58`), reserved words (e.g., `qword`), and operators (e.g., `+`, `[`, and `]`).

Input Embedding. The input representation of the BERT model is a composite embedding, concatenated by three parts: token sequence, segment sequence, and position sequence. The token sequence represents the tokenized instruction sequence. The segment sequence is defined to locate which instruction each token belongs to, while the position sequence denotes an integer sequence encoding the position of each token in the instruction.

Pre-training task. The BERT model proposed two representative pre-training tasks, that is, masked language model (MLM) and next sentence prediction (NSP). MLM will train the model to predict masked tokens in the input sequence, while NSP will train the model to predict the next sentence according to the former one. Liu *et al.* [27] found that the BERT model pre-trained only with MLM can outperform that pre-trained with both tasks. Therefore, we adopt the MLM task to train the BERT model.

In addition to the MLM task, there are also dedicated training tasks proposed for learning assembly code semantics: Context window prediction (CWP) [24] and Def-use prediction (DUP) [21]. As shown in top boxes of Figure 5, the CWP task is to predict if two given instructions co-occur within a context window, where the window size is preset. The DUP task is to determine the relative order of two given instructions. The CWP can capture the control flow information between instructions, while the DUP can grasp the data dependency information across instructions. In this paper, we choose the MLM, CWP, and DUP tasks to train our BERT-based embedding model and then generate the node embedding.

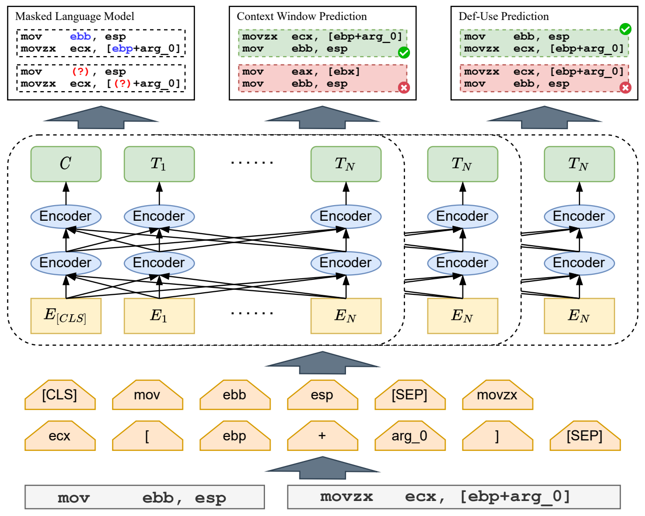


Figure 5: The pretraining of BERT-based embedding model.

B. Edge Embedding

Edge embedding is used to reflect the connectivity between two nodes. The edges in the graph representation involve three types of relationships, i.e., the control flow, control dependency, and data dependency. The edge type refers to which type of connections it belongs to. For example, $[0, 0, 1]$ means the edge represents the DDG, rather than CFG or CDG. Note that there may be multiple edges (e.g., one from CDG and one from DDG) between two nodes. Therefore, the edge embedding is designed as a 3-dimensional binary vector.

3.4 Graph Representation Learning

After converting the graph representation into the embedding format, BinGo will feed the embedding matrices into a detection model, which can provide a powerful capability of graph understanding by the graph neural networks.

As shown in Phase IV of Figure 2, our graph model adopts a siamese structure consisting of two graph convolution networks. Such a structure is conducive to processing pre-patch and post-patch graph representations separately, which can emphasize the differences caused by patches while maintaining similar contexts before and after patches. Each graph network is stacked with the convolution layers and the pooling layers.

As displayed in Figure 6, the graph convolution layer is a variant of the normal convolution layer in the CNN networks, updating the node embeddings by the message propagation among the neighboring nodes. Note that, we limit the number of convolution layers to three because more convolution will cause the graph over-smoothing issue [7], which means excessive convolution may smooth the node attributes in the graph so that their information is permanently lost. Behind the 3-layer graph convolution, we use a graph pooling layer to aggregate all attributes to get the graph embedding. Finally, a binary classifier constructed by multiple-layer perceptron (MLP) is utilized to convert the graph embeddings into predicted labels.

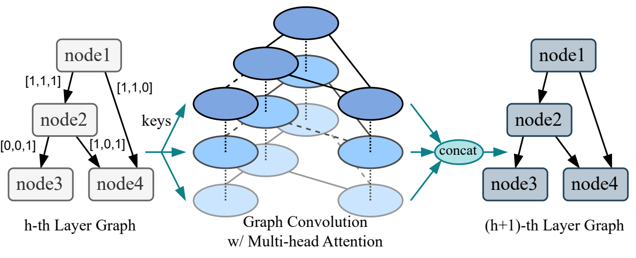


Figure 6: Graph convolutional layer w/ multi-head attention.

Multi-Attributed Graph Convolution. The CPG is a composite graph with diverse edge attributes, representing different connections in CFG, CDG, and DDG. Therefore, we adopt the multi-attributed graph convolution mechanism to process multiple edge types. That is, the graph convolution layer has three channels to process the edge embeddings from different subgraphs.

When the convolution layer updates the node embedding, it collects the embeddings from neighborhood nodes with different subgraph connections. In other words, the graph connection in each convolutional channel depends on the graph type, respectively. Therefore, we train different weights for the each channel of subgraph to learn semantics from different connection relations. Formally, the convolution update can be formalized as follows:

$$\boldsymbol{N}^{(h+1)} = \|_{k=1}^{K} \boldsymbol{\sigma} \left(\left(\boldsymbol{A}^{(k)} + \boldsymbol{I} \right) \cdot \boldsymbol{N}^{(h)} \cdot \boldsymbol{W}_h^{(k)} \right), \quad (2)$$

where $\boldsymbol{A}^{(k)}$ is the adjacency matrix of the k-th subgraph, and I is an identity matrix with the same size of $\boldsymbol{A}^{(k)}$. $\boldsymbol{N}^{(h)}$ denotes the node embedding 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 will be trained via the backpropagation. K is the total number of subgraphs and $\boldsymbol{\sigma}$ is the activation function. Finally, the updated node embeddings from different subgraphs will be aggregated to the convolution result $\boldsymbol{X}^{(h+1)}$ by the vector concatenation. The adjacency matrix $\boldsymbol{A}^{(k)}$ can be thought of as a filter that provides individual attention for each subgraph to train convolution parameters.

Security Patch Classifier. Behind the convolution layer, a pooling layer is used to reduce the embedding dimension. Then, the graph embeddings from both GCN branches are concatenated as the input embedding for the final classifier. Also, 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 fully connected network is built to transform the graph embedding into a binary probability output (p_0, p_1), where $p_0 + p_1 = 1$. The higher probability in the output indicates that a patch instance falls into the category of security/non-security patch.

4 IMPLEMENTATION

In this section, we first introduce how to build the binary patch dataset from a set of pre-collected source code patches. Then, we discuss the implementation of four phases in BinGo, including patch data pre-processing, graph extraction, embedding generation, and graph representation learning.

4.1 Building Binary Patch Dataset

We build the binary patch dataset based on a pre-collected source code patch dataset named PatchDB [46]. We obtain the patch information from the PatchDB and download the corresponding source code files from the GitHub repositories. In this paper, the most challenging part is to compile all patch-related source code files from different software repositories, which require different compile commands and various suitable building environments. We manually prepare the building environments for different software in isolated Docker containers to avoid dependence conflicts, which can provide the build logs. Then, we use a script to automatically extract patch-related source code files from GitHub repositories, and select the correct compile commands and dependent files from the build logs. For versatile evaluation of BinGo on this binary dataset, we leverage two well-known compilers (i.e., Clang and GCC) to compile these programs into binaries with different optimization levels (i.e., `O0`, `O1`, `O2`, `O3`, and `Os`), respectively. Furthermore, we generate the LLVM IR and the assembly code for each patch to facilitate subsequent analysis work.

Compilation Command Database Preparation. Given a source code file for a C program, we need to compile the program based on two types of information, i.e., the dependent files and compiler configuration. The dependent files include the header files and linked source files, while the compiler configuration refers to the options in the compile command line. It is not trivial to seek such complicated information automatically, due to three main challenges. First, the dependent files usually exist at different locations (e.g., *./deps/lua/src*, *./src/lib/openjp2*, *./eglib/src*); thus, complex path relations are difficult to untangle in a general way. Second, some dependent files are generated during the building process, rather than exist before compilation (e.g., *config.h*, *kdb_ldap.h*, *asm/linkage.h*). Third, the format of configuration scripts for automated compilation tools can be largely different (e.g., *autoreconf*, *autogen.sh*).

To solve the above challenges, we propose an automatic extraction system to extract complete compile commands from the build logs. We first manually build the software to collect the build logs. Note that different versions of the same software may also have different dependencies and compilation configurations. Thus, we build isolated Docker containers for each version to avoid dependence conflicts. The build log is parsed into a queryable JSON format. For each software, we build a compile command database to manage the compilation information for each source code file.

Compilation Target. With different compilation configurations, the same source code can even generate diverse assembly code, as demonstrated in Section 2.2. To evaluate if BinGo can handle the changes on compilation configurations, we leverage Clang [8] and GCC [15] with different optimization levels to compile the source code. However, using different compilers and optimization levels means changing the original build environments, which may cause compilation failure. For instance, the earlier versions of Linux kernel can not be compiled with Clang [44], due to the incompatibility with the LLVM toolchain. To solve this issue, we choose the `allyesconfig` [33] to cover as many source code files as possible. The basic idea is to compile the Linux source code module by module, while a module will be skipped if it cannot be compiled successfully. The `allyesconfig` totally includes 16,599 modules,

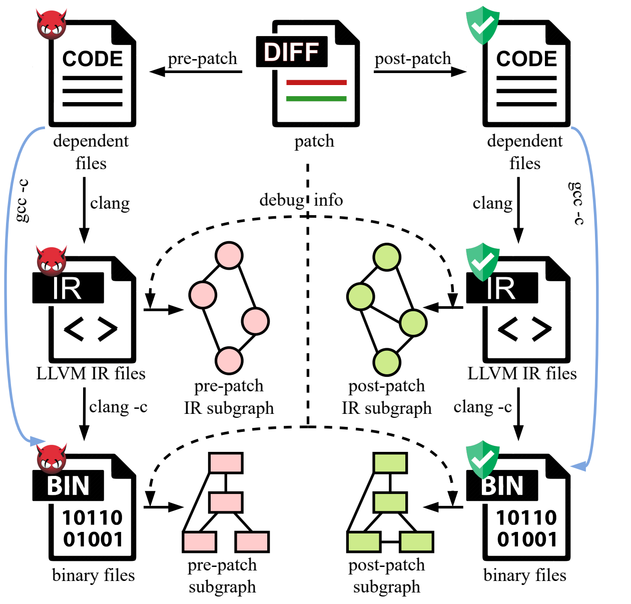


Figure 7: The workflow of binary patch generation via Clang or GCC.

among which we successfully generate LLVM IR for 16,514 modules. Furthermore, the "-g" option is used to generate debug information, which is useful to investigate the code lines and to further locate the patched assembly code snippets.

Automatic Binary Patch Generation. The workflow of binary patch generation is shown in Figure 7. Given the patch information, we can locate the pre-patch and post-patch software versions and obtain patch-related source code files that contain the changed code lines. For the sake of efficiency, we only compile the patch-related source code files. According to the version information and the source code filenames, we can query the corresponding compile commands from the compilation command database for the pre-patch and post-patch source code files.

Clang is able to translate the source code into the LLVM IR code and directly perform optimization and program analysis according to the modular LLVM toolchain. Therefore, in this work, we use Clang to first generate the IR code and then compile it into the binary code. However, different from the structure of Clang, the front-end, middle-end, and back-end of GCC are coupled with each other. GCC cannot directly perform optimization on the IR code. Thus, we use GCC to compile the source code directly into the binary code.

To ensure that the binary code compiled by GCC can be converted into the same assembly and IR format as that compiled by Clang, we also use the LLVM toolchain to disassemble the binary compiled by GCC and lift it into LLVM IR code. As demonstrated in Section 3.1, we locate the patch-related basic blocks in IR and assembly code using the static analysis passes provided by LLVM [3, 50]. Also, we provide the CFG and the CPG-based graph features, which are useful in not only security patch detection but also other patch analysis tasks, such as vulnerability detection, patch presence, and hot patch generation.

Table 1: The statistic of the dataset used to evaluate BinGo.

Patch Type	#Patches	#Funcs	#Nodes		#Edges	
			Avg	Max	Avg	Max
security	1,278	10,630	22.81	840	107.7	7,012
non-security	1,620	14,410	30.38	1,588	150.19	16,933

Composition of Dataset. We retrieve target source code programs from Linux kernel and select 1,278 security patches (fixing CVE vulnerabilities) and 1,620 non-security patches labeled by the PatchDB [46]. PatchDB obtains the security/non-security patch information from NVD and GitHub repositories. For this dataset, we focus on the single-purpose patches whose commit message only describes one issue, e.g., fixing a vulnerability or adding a new feature. All these patches cover hundreds of different Linux kernel versions. For the sake of efficiency, we only collect build logs for the major version of the Linux kernel (e.g., v3.0.0 and v5.8.0). The building environments of the minor versions only change slightly compared with the neighboring major version, thus we can use the same building environment for consecutive minor versions (e.g., v5.6-rc1 and v5.6-rc2).

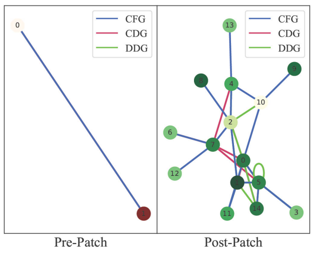
For each patch, we compile the pre-patch and post-patch versions with GCC and Clang under five different optimization levels (i.e., `O0`, `O1`, `O2`, `O3`, and `Os`). Because a patch may change code across multiple functions, a patch may be related to multiple graphs. The statistic information of generated patches is shown in Table 1. Note that, the division ratio between the training set and the test set is 8:2 in the subsequent experiments.

4.2 System Implementation

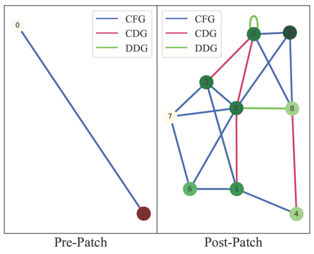
Patched Block Extraction. For the training data in our self-built benchmark, we precisely locate patch-related assembly blocks via mapping relations within debug information. For the testing data, we utilize the modified DeepBinDiff to collect patch-related blocks. We first aggregate basic blocks into functions and then filter similar functions according to the cosine distance of function embeddings. Next, we filter similar basic blocks in the remaining functions, which are identified patch-related blocks. To evaluate BinGo, we totally extract 1,332,031 basic blocks from 2,898 patches, whose statistical information is displayed in Table 1.

Graph Extraction. We leverage the *angr* framework [5] to implement the graph slicing algorithm. To extract the internal graph representation, we first build the CFG to connect all patch-related nodes. Then, we search all relevant external nodes among all three subgraphs, starting from the patch-related nodes. To reduce the graph size and exclude relatively irrelevant nodes, we empirically set the node searching within 2 hops. Also, we set the time limit as 15 minutes for each graph generation to prevent path explosion. Finally, all subgraphs are merged into a CPG. In total, we generate 6,457,351 edges in total for both security and non-security patches.

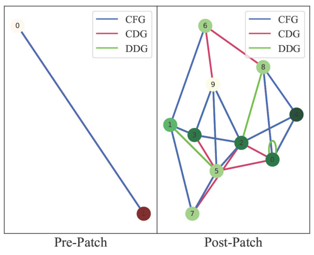
After extracting patch-related blocks and building the graph representation, we find that the graph size of security patches is more likely to be smaller than that of non-security ones, as shown in Table 1. This observation is consistent with the findings in previous



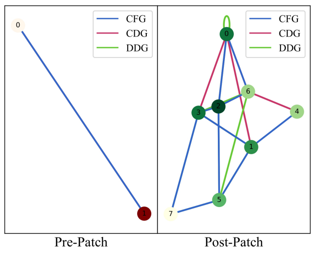
(a) O0



(b) O1



(c) O2 (O3 is the same in this case)



(d) Os

Figure 8: The generated twin graphs using GCC with different optimization levels (CVE-2011-1581).

works [43, 57], which show there are usually fewer code changes in the security patches than those in non-security patches. Besides, we can also confirm that the binary code characteristics, i.e., the graph representations may vary with different compilers and optimization levels, even for the same patch. One example (CVE-2011-1581) is shown in Figure 8.

Embedding Generation. The BERT-based embedding model is implemented using *pytorch* 1.13 and *BERT-pythorch* 0.0.1a4. We set the model structure with 12 encoder layers, each of which contains 8 attention headers. To cooperate with the input size of the subsequent graph model, we also resize the embedding dimension of the BERT model to 128. We adopt a BERT model pre-trained on binary datasets with various compilers and optimization levels [34]. Based on the pre-trained model, we fine-tune the model to fit our dataset in this paper, which is compiled from the Linux kernel. The BERT-based model generates the node embedding, while the edge embedding is a vector composed of the CFG, CDG, and DDG connections. Finally, we integrate the node and edge embeddings together into a unified numeric representation.

Graph Learning. The graph learning program is implemented with Python 3.9.16 while the neural network model is designed with *pytorch* 1.13 and *PyG* 2.2. We also design a *TwinGraph* structure to store and process the pre-patch and post-patch graphs at the same time. In the training phase, the batch size is set to be 128. We use Adam optimizer with β_1 of 0.9, β_2 of 0.99, and a learning rate of 0.001. To prevent the overfitting issue in graph classification, we set a large dropout ratio of 0.5 in the training phase. We also use the cross-entropy loss for the gradient descent. The training set contains 20,031 graph pairs and the testing set contains 5,008 graph pairs; i.e., the training ratio is set to 0.8. The training set and testing set do not contain overlapping commits. The max iteration epoch is 1,000 while the model loss becomes stable.

Table 2: The performance comparison of BinGo and two baseline models for security patch detection.

Model	Dataset	General Metric		Specific Metric	
		Accuracy	F1-score	FN rate	FP rate
PatchRNN [47]	Source	70.86%	0.379	72.97%	7.70%
GraphSPD [43]	Source	85.28%	0.557	56.51%	5.05%
BinGo	Binary	80.77%	0.759	29.15%	11.82%

5 EVALUATION

In this section, we evaluate the performance of BinGo on our binary patch benchmark. We compare the accuracy of BinGo with the state-of-art works on the security patch detection task. Meanwhile, we evaluate the robustness of BinGo under different compilation configurations, i.e., compilers and optimization levels.

5.1 Experiment Setup

Runtime Environments. The dataset generation, patch data preprocessing, graph extraction, and embedding generation are deployed in a Linux server with Intel Xeon E5-2650 @ 2.30 GHz and 512 GB memory, running Ubuntu 20.04 LTS. The GCN-based identification model is carried out in the Ubuntu 22.04 LTS environment running in Intel Xeon Gold 5122 with 3.60 GHz CPU and 64 GB RAM. The model training is conducted by one NVIDIA RTX 2080 Ti GPU of 11 GB memory with CUDA 11.7.

Evaluation Metrics. Security patch detection is a classification task, so we use both general metrics and specific 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. Specific metrics are used to evaluate the practicality of the detection system, including the false-negative rate (FNR) and false-positive rate (FPR).

5.2 Performance on Security Patch Detection

We conduct a series of experiments to answer the following questions.

- **RQ1:** Does BinGo outperform the existing state-of-the-art works?
- **RQ2:** Is BinGo robust on binary patches across different compilers?
- **RQ3:** Is BinGo robust on binary patches across different optimization levels?

RQ1: Does BinGo outperform the existing state-of-the-art works? As shown in Table 2, the BinGo system can achieve up to 80.77% with an F1-score of 0.759 on the binary patch dataset. We compare BinGo directly with the state-of-the-art source code level detection systems PatchRNN [47] and GraphSPD [43]. Note that, we retrained PatchRNN and GraphSPD based on the Linux kernel part of PatchDB [46]. Therefore, the experimental results in Table 2 are comparable.

We can observe that the accuracy of BinGo is slightly inferior to that of GraphSPD, yet it surpasses PatchRNN. Specifically, the BinGo outperforms PatchRNN by 9.91% of accuracy and 0.380 of

Table 3: The performance of BinGo across different compilers and optimization levels.

Dataset		General Metrics		Specific Metrics	
		Accuracy	F1-score	FN rate	FP rate
GCC	O0	82.2%	0.78	30.4%	7.6%
	O1	68.5%	0.62	43.2%	22.0%
	O2	83.0%	0.78	26.1%	10.6%
	O3	81.9%	0.78	27.1%	10.6%
	Os	66.3%	0.54	49.7%	22.9%
	Total	76.4%	0.70	36.0%	14.1%
Clang	O0	89.1%	0.88	14.8%	7.5%
	O1	86.3%	0.83	20.0%	9.3%
	O2	94.6%	0.94	9.9%	1.6%
	O3	95.1%	0.95	7.5%	2.5%
	Os	90.5%	0.89	16.3%	4.2%
	Total	90.3%	0.88	14.7%	5.7%

F1-score. Due to the interference caused by the compilation process, the accuracy of BinGo is 4.51% lower than GraphSPD. However, the BinGo model achieves a higher F1-score compared to state-of-the-art graph models at the source code level, with an improvement of 0.202. The much higher F1-score indicates that BinGo has better precision and recall performance.

To investigate such a performance improvement, we can further compare BinGo with PatchRNN/GraphSPD from the perspective of feature construction and model structure. PatchRNN adopts a sequential embedding model, similar to word2vec [2], to convert the code changes in patches into an embedding representation and uses the twin-structured RNN model as a classifier. Similar to BinGo, GraphSPD constructs the CPGs of both pre-patch and post-patch code; however, GraphSPD merges them into one graph and does not use the sequential embedding model to extract the code semantics. Also, GraphSPD utilizes the GCN to learn the merged patch graph representation and to identify the security patches. In contrast, BinGo has the advantages of both methods. From the perspective of feature representation, BinGo uses the BERT-based embeddings as node attributes in the CPG representation. In addition, BinGo adopts the *TwinGraph* structure to reserve the features of both pre-patch and post-patch graphs. From the perspective of model structure, BinGo adopts the siamese network architecture (same as PatchRNN) to support the pre-patch and post-patch inputs simultaneously; however, each branch of siamese network leverages the GCN structure due to the graph representation. From the experimental results, the feature fusion and the *TwinGraph* structure play a critical role in BinGo by capturing more enriched syntax and semantic features. Although the security patch detection in binary code is more complex than that in source code, BinGo still achieves considerable accuracy and F1-score.

RQ2: Is BinGo robust on binary patches across different compilers? To better understand the impact of compilation on security patch detection, we show the performance of BinGo under various compilers and optimization levels in Table 3. From the results, we notice that BinGo performs much better on the patches compiled by Clang than those compiled by GCC, no matter under which optimization level. Specifically, the overall accuracy of BinGo under Clang is 13.9% higher than that under GGC; meanwhile, the F1-score under Clang is 0.18 higher than that under GCC.

Another important observation is that the number of patches successfully compiled by Clang is far less than that successfully compiled by GCC, due to the high failure rate of compilation. As shown in Figure 9, the ratio of patches compiled by Clang and GCC is 1:2.7. We maintain the same sample ratio for Clang and GCC in both training and validation phases, which means BinGo trains on fewer patches compiled by Clang but learns more salient semantics for better recognition performance.

Such a counter-intuitive result is caused by the inherent mechanism of Clang and the code analysis toolchain (i.e., the LLVM toolchain). On the one hand, the binary patches will be decomposed into more basic blocks due to the compilation mechanism of Clang, hence containing more data dependencies and control dependencies. We analyze the number of nodes and edges in the CPGs extracted by BinGo over different compilation configurations. In Figure 10, we count the average number of nodes and edges in both the pre-patch and post-patch graphs. We find that BinGo can extract more nodes and edges for the patches compiled by Clang. Also, with the compilation of Clang, the graph size difference between security patches and non-security ones is more significant. This means, from the perspective of graph representation, the patches compiled by Clang are more distinguishable that those compiled by GCC.

On the other hand, we notice that compared with the patches compiled by Clang, the patches compiled by GCC may suffer from the imprecise issue during the disassembly, basic block extraction, and CPG construction. This is because the analysis tools used in this paper are all based on the LLVM toolchain (as mentioned in Section 4.1). Because Clang belongs to the LLVM toolchain, BinGo has better compatibility over the patches compiled by Clang. Previous works [4, 16] also introduce the unreliability of reverse engineering tools.

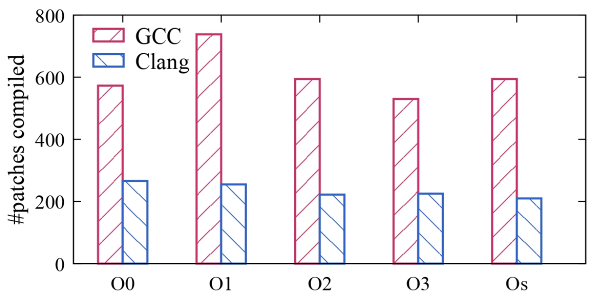


Figure 9: The number of successfully compiled binary patches across compilers and optimization levels.

RQ3: Is BinGo robust on binary patches across different optimization levels? Table 3 shows the performance of BinGo over different optimization levels. Compiled by Clang, BinGo can achieve better accuracy and F1-score on the patches with higher optimization levels (e.g., O2 and O3); however, the performance of BinGo on low optimization levels (e.g., `O0` and `O1`) is relatively poor, with the accuracy lower than 90% and an F1-score of 0.88. Also, BinGo can achieve moderate performance on the patches optimized with `Os`. Nevertheless, the performance gap of BinGo over different optimization levels does not exceed 10%. Compared to Clang, the performance over different optimization levels maintains a similar trend on GCC-compiled patches. However, we also notice that the performance of `Os`-optimized patches drops dramatically to 66.3% if compiled by GCC. In this case, the performance gap reaches 16.7%, compared with both high and low optimization levels.

To investigate the performance trend across optimization levels, we first observe the graph features extracted by BinGo under different optimization levels. As shown in Figure 10, we notice that BinGo can extract more nodes and edges for the patches with high optimization levels, while the size of generated graphs with low optimization levels will be relatively small. Among different optimization levels, BinGo extracts the fewest nodes and edges for the `O1`-optimized patches. The distribution is consistent with the general performance trend of `O0`, `O1`, `O2`, and `O3` for both Clang-compiled and GCC-compiled patches. Therefore, we can conclude that the higher the optimization level, the better the identification performance of BinGo. Moreover, after comparing the number of nodes and edges for security patches and non-security patches, we find that the patches compiled by GCC with `Os` have the most subtle differences (as shown in Figure 10(a) and 10(c)). This also explains why the performance of BinGo drops drastically under this scenario.

Second, we find that GCC and Clang support different sets of optimization flags. For the GCC, `Os` selects optimization options in `O2` that do not increase the object file size; while for Clang, `Os` is a combination of `O2` with extra optimizations to reduce code size. Therefore, compared with `O2`, there are even fewer optimization items in `Os` if compiled by GCC. This is another possible reason for the performance of BinGo in this case. In practice, `O2` is the default optimization level used in most scenarios, while `Os` is merely used in compilation environments with tight memory resources. Therefore, the performance of BinGo under the GCC compiler with `Os` optimization is acceptable.

5.3 False Positive and False Negative

In the security patch detection task, a false positive (FP) means that a real non-security patch is misidentified as a security patch. A false negative (FN) represents that a real security patch was missed out. From Table 2, we can observe that BinGo achieves the best FNR of 29.15%, which is only half of the FNR in these two baseline approaches. Also, the FPR (11.82%) of BinGo is worse than that of PatchRNN (7.70%) and that of GraphSPD (5.05%). That means the users may get twice as many false security patch alerts when using BinGo to detect binary patches. However, the FPR is acceptable considering the extreme imbalance between non-security and security patches in practice. The security patches only account for 6-10% in open source software (OSS) [46]. Therefore, both FPR and FNR of BinGo are better than those of the previous methods in general, which is consistent with the results in the F1-score.

Next, we compare the specific performance (FNR and FPR) of BinGo across compilers and optimization levels. We observe that the

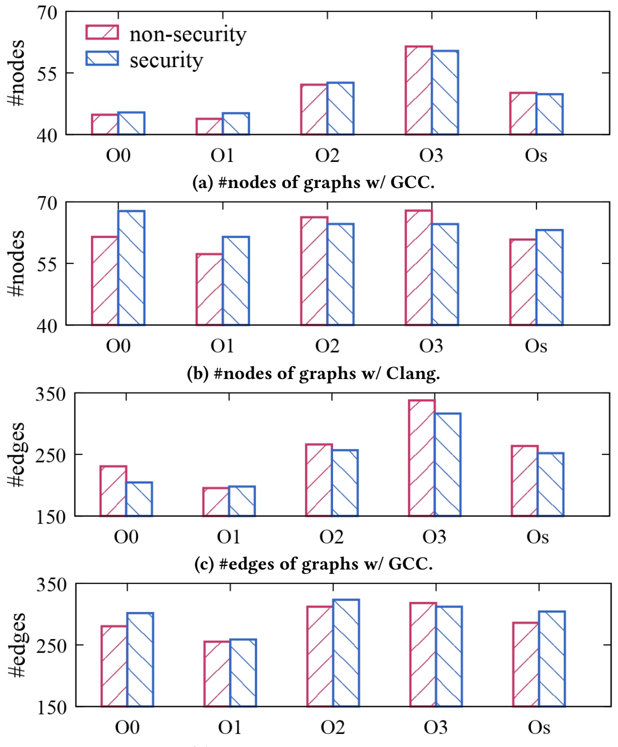


Figure 10: The graph size of patches with different compilers and optimization levels.

trend of FPR and FNR of BinGo is consistent with that of F1-score. It is worth noting that even in the worst case (under GCC with `Os` optimization), our BinGo still outperforms previous methods. Considering that the comparison in this paper is between source code patches and binary patches, the actual performance of BinGo can even advance more against the previous methods.

6 DISCUSSION

The evaluation results demonstrate that BinGo can effectively identify the security patches in binary code. Although the composition of binary code changes with the compiler and optimization level, BinGo is robust to eliminate such noisy impact caused by compilation. BinGo is able to identify hidden security patches released by the software vendors or from any binary code diffing between neighboring software versions. BinGo can help developers or IT operators discover binary with security patches, and update them selectively, so as to maintain the stability and security of functions. Besides, our survey found that 58.55% (469/801) of security patches appear in the same function. If we fix a bug in the binary, BinGo can locate the function to be modified, and the developer can replace only the relevant function. This reduces the complexity of manual repair.

However, we also notice that BinGo still has some limitations, so corresponding further work can be taken into consideration. First, BinGo does not consider a scenario that two patches coexist in one binary version, in which we need to split the tangled patches.

Existing work [37, 42] has achieved patch untangling at the source code level, but there is no similar work at the binary level so far. Therefore, we leave the untangling solution for binary patches in our future work. Second, BinGo does not consider the semantics of the callee function when constructing the CPG representation. That is, if the patch involves a function call, the CPG only involves the basic blocks in the caller function, but not those in the callee. However, if all callee procedures are included, the CPG will be too large to be learned. Therefore, in further work, the trade-offs between more semantics and compact graphs are required. Additionally, security patches can be further categorized, which may be helpful to reduce false negative rates. In our dataset, the category distribution of security patches is not uniform, e.g., the If-then-else structures account for more than 70% of security patches. Accordingly, when identifying security patches of a minority category, the false negative rate may be high. With more security patch types, BinGo can be well-trained to better classify security patches.

7 RELATED WORK

Code Representation. Regardless of whether it is source code or binary code, extracting the code representation is the first step in the program analysis. The graph representation is the natural way to represent the intrinsic program flow structure [1, 53, 55]. By defining different edges and nodes, graphs can be utilized for diverse program analysis tasks. For example, Yamaguchi *et al.* [53] first defined the code property graph (CPG) by integrating multiple types of graphs to detect the vulnerabilities. Ji *et al.* [19] encoded multiple statistical features into the attributed control flow graph (ACFG) to identify the compilation provenance. Inspired by the popularity of the large language model (LLM), researchers employed self-supervised sequence embedding models [10, 22, 36] to capture the textual semantics from code [14, 24, 26]. In industry, the LLM-based embedding model has achieved good performance in code generation and debugging tasks, such as the Codex [32], and ChatGPT [31].

Patch Analysis. Due to the development of open-source software (OSS) and program reuse, patch analysis has become especially important for the security of the software supply chain [23, 40]. Wang *et al.* [46] collected massive patch information from the NVD database and GitHub and provided a benchmark for patch analysis. Li *et al.* [23] empirically studied the syntax structure of security patches, revealing multiple significant behaviors. Patch analysis facilitates several downstream tasks, such as vulnerability detection and automated program repair. On the one hand, security patch detection is the mirror problem of vulnerability detection [9, 20, 54, 56]. Detecting a security patch indicates the corresponding vulnerability has been fixed. Tian *et al.* [41] utilized textual and code features of source code to detect bug-fixing patches in Linux. Zhang *et al.* [54] proposed the patch presence test for binaries, which maps the patch patterns in source code changes and checks the presence of such patterns in the binary code. On the other hand, researchers analyze the pattern and structure of security patches and imitate them to generate patches automatically [11, 30]. Aravind *et al.* [28] defined the safe patch that does not disrupt the intended functionality of the program. Such a safe patch can be propagated in the software supply chain. Xu *et al.* [52] proposed Vulmet which can automatically generate hot patches for Android via learning patch semantics. Wang *et al.* [48] further used the random forest with extracted patch features to classify security patches into specific vulnerability types.

Security Patch Identification. All the above works have a common premise that the object patch must be security-related. Distinguishing between security patches and non-security patches can warrant such an assumption. Researchers constructed syntactic and semantic features from the source code and commit messages, then employed machine learning (ML) or even deep learning (DL) based classifiers to distinguish security patches from other patches [17]. For example, PatchRNN [47] and SPI [57] identify security patches with RNN models. Wang *et al.* [43] developed graphSPD, which conducted multiple-attribute graph representation based on the commit information and exploited the graph learning model to detect security patches. Regarding binary patch detection, to the best of our knowledge, there is no existing work that focuses on distinguishing security patches from non-security ones. Thus, BinGo is the first approach that focuses on detecting binary security patches.

8 CONCLUSION

In this paper, we propose BinGo, a new end-to-end binary patch identification system. BinGo can be performed in two steps. BinGo first leverages the code property graph and LM-based embedding model to encode the semantics of binary patches, and then distinguishes the security patches via a Siamese-structured GCN model. BinGo can help users and developers select critical security patches from unknown binary patches to ensure software security and functional stability. To train and evaluate BinGo, we also propose an automatic approach to generate a binary patch dataset according to known patch information in the source code. We implement a prototype of BinGo and conduct experiments to evaluate the effectiveness and robustness of BinGo. The experimental results show that BinGo can achieve great performance (80.77% accuracy and 0.759 F1-score) for identifying security patches. Binary patches can be harder to detect compared to source code patches due to various compilation configurations. However, BinGo outperforms the state-of-art solutions that focus on identifying patches in source code, and exhibits good robustness to patches across compilers and optimization levels. In addition, the experimental results show that BinGo has fewer false alarms, which means that BinGo rarely misses or misidentifies security patches.

ACKNOWLEDGMENTS

This research is partially supported by ONR grant N00014-23-1-2122, NSF grant CNS-1822094, and NSFC Grant 62132011.

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. arXiv:1711.00740 [cs.LG]

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

[3] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. 2020. BinRec: Dynamic Binary Lifting and Recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)* (Heraklion, Greece). Association for Computing Machinery,

New York, NY, USA, Article 36, 16 pages. https://doi.org/10.1145/3342195.3387550

[4] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, USA, 583–600.

[5] Angr. 2021. A python framework for analyzing binaries. http://angr.io/.

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

[7] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. 2020. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. Association for the Advancement of Artificial Intelligence, New York, USA, 3438–3445.

[8] Clang Team. 2020. clang - the Clang C, C++, and Objective-C compiler. https://clang.llvm.org/docs/CommandGuide/clang.html.

[9] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Online, 1147–1164. https://www.usenix.org/conference/usenixsecurity20/presentation/dai

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[11] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries. In *26th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, San Diego, CA, USA, 15 pages. https://dx.doi.org/10.14722/ndss.2019.23126

[12] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DEEPBINDIFF: Learning Program-Wide Code Representations for Binary Diffing. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20)*. Internet Society, SanDiego,CA,USA, 16 pages. https://dx.doi.org/10.14722/ndss.2020.24311

[13] Nitzan Farhi, Noam Koenigstein, and Yuval Shavitt. 2023. Detecting Security Patches via Behavioral Data in Code Repositories. arXiv:2302.02112 [cs.CR]

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics (EMNLP 2020)*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[15] GCC team. 2018. Options That Control Optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[16] Xu He, Shu Wang, Yunlong Xing, Pengbin Feng, Haining Wang, Qi Li, Songqing Chen, and Kun Sun. 2022. BinProv: Binary Code Provenance Identification without Disassembly. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'22)*. Association for Computing Machinery, New York, NY, USA, 350–363. https://doi.org/10.1145/3545948.3545956

[17] Thong Hoang, Julia Lawall, Richard J. Oentaryo, Yuan Tian, and David Lo. 2019. PatchNet: A Tool for Deep Patch Classification. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019)*. IEEE Press, Montreal, Canada, 83–86. https://doi.org/10.1109/ICSE-Companion.2019.00044

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

[19] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. Vestige: Identifying Binary Code Provenance for Vulnerability Detection. In *Applied Cryptography and Network Security (ACNS 2021)*. Springer International Publishing, Cham, 287–310.

[20] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-Based Patch Presence Testing for Downstream Kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1149–1163. https://doi.org/10.1145/3372297.3417240

[21] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *8th International Conference on Learning Representations, ICLR 2020*. OpenReview.net, Addis Ababa, Ethiopia, 16 pages. https://openreview.net/forum?id=H1eA7AEtvS

[22] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. arXiv:1910.13461 [cs.CL]

[23] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 2201–2215.

[24] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. ACM, New York, NY, USA, 3236–3251.

[25] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*. PMLR, LongBeach,CA, USA, 3835–3845.

[26] Shangqing Liu, Yanzhou Li, Xiaofei Xie, and Yang Liu. 2023. CommitBART: A Large Pre-trained Model for GitHub Commits. arXiv:2208.08100 [cs.SE]

[27] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]

[28] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. SPIDER: Enabling Fast Patch Propagation In Related Software Repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1562–1579. https://doi.org/10.1109/SP40000.2020.00038

[29] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. 2021. Modeling functional similarity in source code with graph-based siamese networks. *IEEE Transactions on Software Engineering* 48, 10 (2021), 3771–3789.

[30] Christian Niesler, Sebastian Surminski, and Lucas Davi. 2021. HERA: Hotpatching of Embedded Real-time Applications. In *Proc. of 28th Network and Distributed System Security Symposium (NDSS)*. Internet Society, SanDiego,CA,USA, 16 pages. hhttps://dx.doi.org/10.14722/ndss.2021.24159

[31] OpenAI. 2022. OpenAI ChatGPT. https://openai.com/blog/chatgpt.

[32] OpenAI. 2022. OpenAI Codex. https://openai.com/blog/openai-codex.

[33] Fabio Pagani and Davide Balzarotti. 2021. Autoprofile: Towards automated profile generation for memory analysis. *ACM Transactions on Privacy and Security* 25, 1 (2021), 1–26.

[34] PalmTree. 2021. Pre-trained BERT model. https://drive.google.com/file/d/1yC3M-kVTFWql6hCgM_QCbKtc1PbdVdvp/view.

[35] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. 2021. XDA: Accurate, Robust Disassembly with Transfer Learning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*. Internet Society, USA, 1–18.

[36] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[37] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. 2021. SmartCommit: A Graph-Based Interactive Assistant for Activity-Oriented Commits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)* (Athens, Greece). Association for Computing Machinery, New York, NY, USA, 379–390. https://doi.org/10.1145/3468264.3468551

[38] The DWARF Debugging Standard. 2021. Welcome to the DWARF Debugging Standard Website. http://dwarfstd.org/.

[39] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. 2021. Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. Association for Computing Machinery, New York, NY, USA, 3282–3299.

[40] Synopsys technology. 2023. 2023 Open Source Security and Risk Analysis Report. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html.

[41] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux bug fixing patches. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Zurich, Switzerland, 386–396. https://doi.org/10.1109/ICSE.2012.6227176

[42] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2019. Cora: Decomposing and describing tangled code changes for reviewer. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, San Diego, CA, USA, 1050–1061.

[43] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li. 2023. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics. In *2023 2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 604–621. https://doi.org/10.1109/SP46215.2023.00035

[44] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. Association for Computing Machinery, New York, NY, USA, 1899–1913.

[45] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2019. 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). IEEE, IEEE, Portland, OR, USA, 485–492.

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

[47] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, IEEE, San Francisco, CA, USA, 595–600.

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

[49] Bozhi Wu, Shangqing Liu, Ruitao Feng, Xiaofei Xie, Jingkai Siow, and Shang-Wei Lin. 2022. Enhancing security patch identification by capturing structures in commits. *IEEE Transactions on Dependable and Secure Computing* 20, 6 (2022), 15 pages.

[50] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, USA, 1–18.

[51] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, Buenos Aires, Argentina, 462–472. https://doi.org/10.1109/ICSE.2017.49

[52] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. 2020. Automatic Hot Patch Generation for Android Kernels. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, New Yord, NY, USA, 2397–2414. https://www.usenix.org/conference/usenixsecurity20/presentation/xu

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

[54] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *27th USENIX Security Symposium (USENIX Security)*. USENIX Association, Baltimore, MD, 887–902. https://www.usenix.org/conference/usenixsecurity18/presentation/zhang-hang

[55] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to Represent Programs with Heterogeneous Graphs. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 378–389. https://doi.org/10.1145/3524610.3527905

[56] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. 2021. An Investigation of the Android Kernel Patch Ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, New York, NY, USA, 3649–3666. https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-zheng

[57] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. 2021. SPI: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–27.