BinHunter: A Fine-Grained Graph Representation for Localizing Vulnerabilities in Binary Executables*

Sima Arasteh*, Jelena Mirkovic†, Mukund Raghothaman* and Christophe Hauser‡
*Thomas Lord Department of Computer Science
University of Southern California, Los Angeles, CA 90089
Email: {arasteh, raghotha}@usc.edu
†Information Sciences Institute and Thomas Lord Department of Computer Science
University of Southern California, Los Angeles, CA 90089
Email: mirkovic@isi.edu
‡Department of Computer Science
Dartmouth College, Hanover, NH 03755
Email: Christophe.Hauser@dartmouth.edu

***Abstract*—The success of deep learning techniques in diverse fields has prompted research into their application for automatic software vulnerability discovery. The first step in the design of a deep learning based vulnerability detector fundamentally involves selecting an appropriate binary representation. A second challenge arises from the need to automatically localize the vulnerability to specific instructions, so as to allow for better detection and to enable downstream applications such as triage and patching.**

In this paper, we propose BinHunter, an automated tool for vulnerability discovery in binary programs. BinHunter leverages a new graph representation derived from slices of the combined control and data dependency graphs of a binary executable, and can learn code properties by propagating information through the graph edges. This representation enables graph convolutional network (GCN) learning algorithms to both detect and pinpoint the locations of vulnerabilities in binary programs.

We evaluate our approach both using the Juliet test suite and a dataset consisting of historical CVEs from the Debian packages. In both evaluations, we observe that BinHunter is significantly more effective than the baselines: On the Juliet test programs, our model has 6.77%, 26.53%, 24.65% and 41.59% higher true positive rates and 19%, 47.64%, 31.47% and 39.82% lower false positive rates than our baselines respectively (Bin2vec [1], Asm2vec [10], Genius [12] and Jtrans [46]). Furthermore, our model is able to detect 17 of 21 bugs from the Debian dataset, Bin2vec detects 2 bugs, and the remaining three baselines are unable to detect any vulnerabilities at all.

***Index Terms*—Vulnerability discovery, deep learning, graph convolutional networks**

* *This material is based on research sponsored by the National Science Foundation via contracts 2146518, 2107261, and 1815495. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation.*

1. Introduction

The frequent unavailability of source code and the diversity of programming languages used to build software necessitates techniques that can directly detect vulnerabilities from the compiled binary versions of programs. Researchers have explored a range of static and dynamic techniques for vulnerability discovery in binaries.

Among static analysis techniques, signature-based approaches [5], [29], [33], [48] rely on pattern recognition. Since these approaches define specific rules for each vulnerability type, they can detect only certain kinds of vulnerabilities and are sensitive to binary code alterations. More recently, machine learning-based techniques have been proposed that rely on manual feature extraction for vulnerability discovery [11], [12], [15], [17], [20], [47]. In this context, deep learning-based techniques promise to free researchers from manual feature extraction necessary for conventional (non deep-learning-based) methods and have demonstrated promising outcomes across various domains [3], [4], [18], [19], [36], [39], [41].

On the other hand, applying deep learning methods [1], [10], [42], [45] for vulnerability discovery raises other types of questions: How do we represent the program, determine the granularity (statements, basic blocks, functions, etc.) at which detection is performed, and what neural network model do we use? Most function-level techniques [1], [10], [12], [20], [30], [47] build on sequence-based models from natural-language processing (NLP), and do not directly access information about data and control dependencies (for e.g., user-controlled inputs being passed to a database without sanitization) that more directly indicate the presence of vulnerabilities. As a consequence, most of these techniques have focused on tasks such as function similarity or plagiarism detection.

Because vulnerabilities are often confined to specific instructions within an otherwise large function, function-level

techniques are prone to overfitting to the function semantics rather learning the root cause of vulnerabilities. Furthermore, even when function-level detection is successful, manual effort is needed to locate the vulnerable instructions for triage and patching.

We propose BinHunter, a novel graph-based representation of a binary program derived from the sliced program dependence graph (PDG) that contains both data and control dependencies. Our contributions are:

1) We introduce a novel graph representation derived from the PDG, and which encapsulates both data and control dependencies. We show how a graph convolutional network (GCN) directly operating over this graph can learn program properties by propagating information through its edges.
2) We propose a slicing technique to extract subgraphs from the PDG which not only facilitates bug detection, but also helps in pinpointing its root cause.
3) We construct a dataset of CVEs by correlating the database of Debian snapshots with the National Vulnerability Database (NVD). Each of these CVEs includes information about the vulnerability type ("CWE", or the Common Weakness Enumeration), its precise location (including function name, line numbers and affected binary instructions), and versions of the vulnerable and patched function (both at source and binary levels). To the best of our knowledge, this is the first such publicly available dataset.
4) We evaluate BinHunter on this dataset and compare it to a set of four state-of-the-art baselines: Bin2Vec [1], Asm2vec [10], Genius [12] and Jtrans [46]. We show that our system consistently outperforms all four baselines, both on our dataset and when applied to the Juliet test suite [32].

The rest of this paper is organized as follows: We start with a review of related work in Section 2. We then provide the design overview of BinHunter in Section 3. We present the implementation details and the results of our experimental evaluation in Sections 4 and 5 respectively. We then briefly discuss potential limitations of our technique in Section 6.

2. Related Work

Researchers have extensively studied software vulnerability discovery using both static [1], [5], [10]–[12], [17], [29], [30], [33], [45], [47], [48] and dynamic [14], [50], [52] analysis. In this section, we review static analysis for vulnerability detection in *binaries*; for a comprehensive survey of deep learning based vulnerability detection at the level of *source code*, we refer the reader to [7].

2.1. Deterministic Methods

Researchers have developed tools that employ signatures and rules to identify vulnerabilities. Both VulMatch [29] and BinXray [48] derive vulnerability signatures based on differences between vulnerable and patched versions of code. Similarly, UbSym [5] defines specifications to identify potentially vulnerable segments of the program, and then applies targeted symbolic execution to find memory corruption vulnerabilities. Unsurprisingly, these methods are sensitive to code modifications and compiler optimization levels, and they require frequent reformulation of rules for different vulnerabilities and compiler options.

2.2. Conventional Machine Learning Methods

Conventional machine learning methods rely on manually extracting a set of features from binary programs. For instance, Genius [12] introduced the Attributed Control Flow Graph (ACFG), which extracts a statistical and structural feature set from basic blocks. Other techniques, including Gemini [47], BugGraph [20], and Vulseekerpro [15] have also used ACFGs for their models.

Genius used spectral clustering [35] and a graph matching algorithm on ACFGs and calculated the distance between a new sample ACFG from the test set and cluster centroids. To reduce the computational cost of Genius's graph matching, Gemini transformed each ACFG into a vector using the Siamese architecture [6] and the structure2vec [9] model. Meanwhile, Vulseekerpro outperformed Gemini by employing a two-step filtering algorithm. Vulseekerpro is a bug search engine that eliminates functions dissimilar to the target function and selects the top M similar functions. It then reorders the selected functions to identify the top N matches using dynamic analysis.

DiscovER [11] and VDiscover [17] both extract a set of manually identified features from functions. For instance, DiscovER extracts numerical and structural statistics, such as the number of instructions and the size of local variables. Similarly, VDiscover utilizes a combination of static and dynamic properties—such as a set of calls to standard C libraries—to classify functions as vulnerable or benign.

All of the methods mentioned in this section apply machine learning algorithms to a set of manually extracted features. Since manual feature extraction relies on human effort and knowledge, it is time-intensive and imperfect. It also requires new feature engineering as new vulnerabilities arise. In contrast, our approach applies deep learning to automatically craft important features during training. This minimizes the human effort for model development and makes retraining easy to capture new vulnerabilities.

2.3. Deep-Learning-Based Methods

Due to the outstanding performance of deep learning in other domains, researchers have begun to explore its applications to software vulnerability discovery. Two main challenges when using deep learning-based approaches [10], [30] are the choice of representation of the binary program, and the granularity level at which training and testing are performed. We review the history of deep learning based methods from these two aspects, and provide a comparative summary in Table 1.

TABLE 1: Comparison of deep learning based binary vulnerability detectors.

Method	Localization	Code Properties	Model
Asm2vec [10]	No	Ctrl flow	PV-DM
Bin2vec [1]	No	Ctrl flow	GCN
BVdetector [45]	Yes	Ctrl and data deps	word2vec
Zeek [42]	No	Data flow	proc2vec
BinHunter	Yes	Ctrl and data deps	GCN

2.3.1. Granularity of detection. Vulnerabilities typically only affect a subset of a body of a given function in a binary. Based on the granularity of representation, we can distinguish coarse-grained and fine-grained detectors, depending on whether they operate at the level of a whole function or parts of each function. The majority of methods work at function-level granularity, and therefore cannot pinpoint the vulnerability location within the function. Examples include Asm2vec [10], which converts binary functions into vector representations using the PV-DM model [24], and Bin2vec [1].

The lack of fine-grained localization limits their effectiveness in downstream tasks such as triage and patching. Moreover, real-world programs often consist of large functions. In this situation, there is a risk that the classifier is unable to precisely identify the aspects of functions related to vulnerability and mistakenly learns parts of the function semantics instead. We will see evidence of such over-fitting in our ablation study in Section 5.4.

To achieve fine-grained vulnerability detection, researchers leverage program slicing techniques. BVdetector [45] learns vulnerability patterns from program slices derived from calls to API/library functions. Moreover, Zeek [42] converts code fragments into vector inputs. Each code fragment contains all sets of instructions that contribute to the value of a variable. This way, when a vulnerability is detected it can be localized to vicinity of specific API/library calls (BVdetector) or to program segments required for a variable calculation (Zeek). Nevertheless, the main shortcoming in these methods is the chosen program representation (word2vec and proc2vec respectively) cannot reliably capture long-range dependencies among program elements.

2.3.2. Program representation. The second challenge with deep-learning-based approaches is therefore the choice of program representation. Most models represent the binary program as a textual sequence of assembly instructions and rely on sequence-based models such as recurrent neural networks (RNNs) or long short-term memories (LSTMs) to learn classifiers [2], [28].

BVdetector [45] and Schaad et al. [40] use word2vec [8] to transform binary instructions into vectors and learn code representation. In contrast, instruction2vec [25] embeds each assembly instruction into a vector and uses TextCNN [21] to learn both vulnerable and patched functions.

Asm2vec [10] uses the PV-DM [24] model. PV-DM [24] is an extension of the word2vec model and can jointly learn representations of words and paragraphs.

In contrast to purely sequence-based models such LSTMs and BiLSTMs, structured models such as TreeLSTMs and GCNs hold promise in using information derived from program analyzers. For example, the graph embedding used by Bin2vec captures control flow. In contrast, our present model, BinHunter is able to capture both data and control dependencies, and consequently significantly outperforms Bin2vec in our experiments.

3. The Design of BinHunter

We describe our insight and innovations in Section 3.1. BinHunter starts by constructing the program dependence graph (PDG), as described in Section 3.2. It then slices the PDG into smaller code segments, to facilitate localization of vulnerabilities, as described in Section 3.3. An overview of the workflow is shown in Figure 1.

3.1. Insight and Innovations

The central problem that we solve using BinHunter is to identify and localize vulnerabilities within binary executables.

Insight #1: We need both control and data dependencies. Vulnerabilities often arise because a user-controlled input may drive program execution to an unintended path or it may influence the value of some critical variable. Thus, accurate detection of vulnerabilities requires both control and data dependency between binary code statements. Our first innovation resides in a novel graph that captures both forms of information in our representation of binary code, and using this information to learn the vulnerability classifier. As we will show in our experiments, removing either form of information reduces detection accuracy.

Insight #2: We need small code segments for learning. Our goal is to produce suitable segments of binary code for training and classification, so that we can precisely localize a given vulnerability. Many vulnerabilities arise due to only a small subset of the code within a function. Despite this, the vulnerability may be spread across multiple basic blocks in distant portions of the code. These two simultaneous characteristics make vulnerability detection difficult, because it is challenging to *tune* the detector to patterns specific to the vulnerability, rather than to the large body of surrounding code. Our second innovation is our introduction of *PDG slicing*, which greatly improves localization of identified vulnerabilities, when compared to use of whole functions for machine learning. Traditionally, such localization requires a significant manual effort that BinHunter is able to automate. Furthermore, PDG slicing also helps to improve classifier accuracy, as learning occurs over instructions specific to the vulnerability rather than the surrounding code.

3.2. Obtaining the Program Dependence Graph

BinHunter operates on an intermediate representation (IR) of the executable produced by a binary analysis engine. The

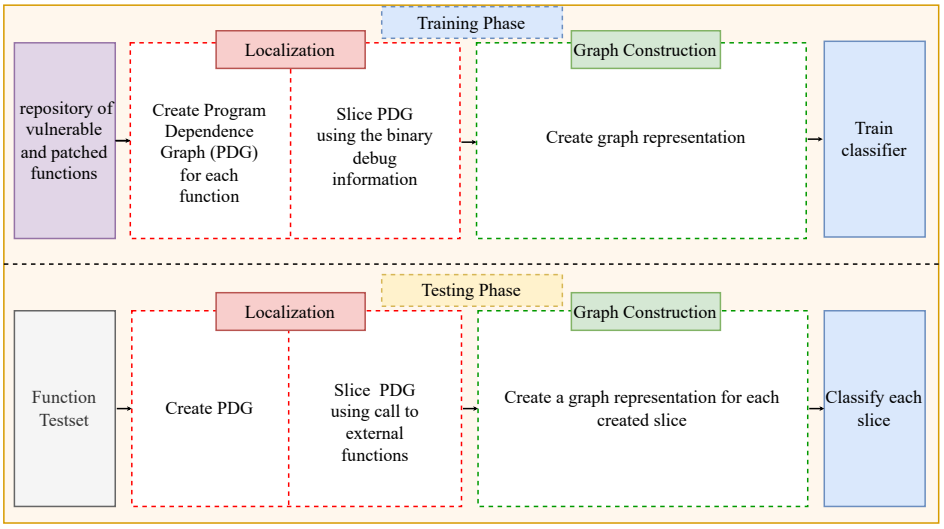

Figure 1: An overview of the BinHunter workflow.

```
0x101265:(unique, 0xef80, 8) COPY (register, 0x28, 8)
0x101265:(register, 0x20, 8) INT_SUB (register, 0x20, 8) , (const, 0x8, 8)
0x101265: ---  STORE (const, 0x1b1, 4) , (register, 0x20, 8) , (unique, 0xef80, 8)
0x101265:(register, 0x20a, 1) COPY (const, 0x0, 1)
```

Figure 2: p-code statements corresponding to the assembly instruction: 0x101265: push rbp.

IR specifies each assembly instruction in terms of its opcode, operands and destination. These operands and destinations may either be registers, constants, memory locations, or intermediate values arising during disassembly. We show an example of this IR in Figure 2.

For formal definitions of control and data dependency graphs (CDG and DDG), we point the reader to the classic reference [13]. Informally, we say that a basic block B_1 in a function is *post-dominated* by another block B_2 if every path from B_1 to the exit node passes through B_2. Subsequently, we say that the basic block B_2 is control-dependent on another basic block B_1 if B_1 is *not* post-dominated by B_2, but there is a path from B_1 to B_2 along which every intermediate block B_3 is post-dominated by B_2. In other words, the result of evaluating block B_1 can determine whether or not B_2 is eventually executed. Binary analysis engines typically either directly provide the CDG or provide APIs to either determine post-domination, from which we can manually construct the CDG.

We construct the DDG using APIs provided by the binary analysis engine. There is a data dependence between two IR statements s_1 and s_2 if s_2 may use the value produced as a result of executing s_1. We note that determining accurate CDGs and DDGs are intractable problems in binary analysis, because binaries often lack complete information about variable sizes, types and possible values, and all this information influences control paths and data flows. In this work we rely on imperfect versions of CDG and DDG, supplied by binary analysis engines. Our evaluation shows that these versions still perform well for vulnerability detection and localization.

Finally, we observe that the CDG and the DDG are produced at different levels of granularity, where the CDG contains edges between basic blocks, and the DDG contains dependencies between different variables in the IR statements. See Figure 3 for an example of this difference in granularity, where control dependencies are shown as edges between basic blocks, and data dependencies are shown using differently colored variable names.

3.3. PDG Slicing

Our goal is to extract subgraphs of the PDG that precisely capture individual vulnerabilities. Having too many instructions in these code segments can cause the model to learn irrelevant patterns from surrounding code, while having too few instructions might fail to capture the data and control dependencies necessary to trigger the root cause and thereby jeopardize classification accuracy. We use different techniques to slice the PDG at training and at test time.

During *training*, we assume that we have access to a large, labeled dataset of vulnerabilities and the corresponding source code for vulnerable and patched binaries. We construct slices that precisely capture vulnerabilities as follows. We start by consulting the DWARF debugging information for the pair of binaries, and obtain a seed set V_0 of vulnerable instructions. We then expand this set of instructions to also include all forward V_f and backward V_b data dependencies within the function being analyzed, resulting in a subgraph of the PDG that precisely captures the vulnerability.

For example, consider the following sequence of instructions:

```
L1: w := a + 5
L2: x := a + 1
L3: z := a + 2
```

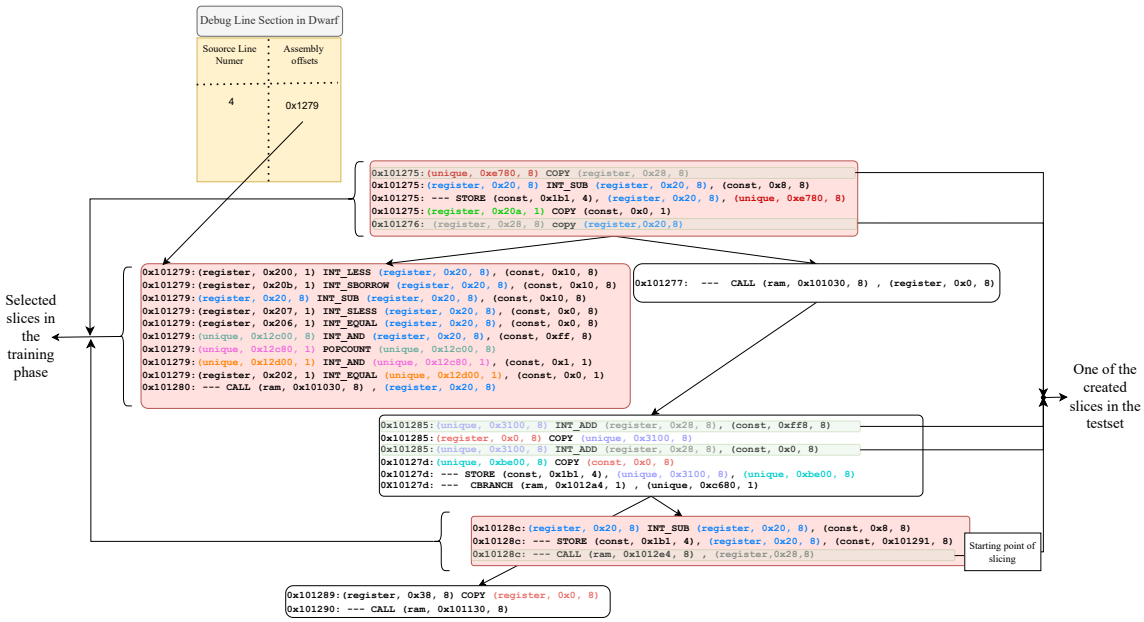

Figure 3: Approaches to slice the PDG at training and test time. Each basic block consists of a sequence of P-code operations, and the edges between basic blocks indicate control dependence edges. We illustrate data dependencies using differently colored varnodes (registers, temporaries, etc.). For instance $(register, 0x20, 8)$ is used in instructions 0x101275, 0x101276, 0x101279, 0x101280, 0x10128c showing the data dependency between these instructions. Training-time slices are extracted from ground truth knowledge of the vulnerability and by performing forward and backward reachability in the data dependency graph. At test time, we start from calls to external functions and similarly perform forward and backward reachability. Four such slices will be extracted from this PDG, starting from assembly offsets 0x1277, 0x1280, 0x128c and 0x1290 respectively. Only the slice starting from 0x128c is shown.

```
L4: y := a + 5
L5: y := w + 3
L6: x := y + z // <--Modified in patch
L7: c := x + 3
L8: d := c + w
```

One may imagine each of these instructions as corresponding to an IR statement, such as:

```
(register, 0x20, 8)
  INT_ADD (register, 0x28, 8),
          (register, 0x30, 8)
```

Furthermore, say that the assignment `L6: x := y + z` was modified in the patch. In this case, $V_0 = \{\texttt{L6}\}$, and $V_b = \{\texttt{L3}, \texttt{L5}\}$. Similarly, V_f is the set of IR statements which are data-dependent on some statement in V_0. In our example, $V_f = \{\texttt{L7}\}$. $V_0 \cup V_f \cup V_b$ may be thought of as the set of instructions relevant to the vulnerability. At the end of the training run, we define n to be the average number of assembly instructions contained in $V_0 \cup V_f \cup V_b$. We visualize this process in Figure 3.

On the other hand, this approach would not work at *test time*, because we would not have a priori knowledge of vulnerability locations and stripped binaries would not have any external information for slicing. We instead rely on the observation previously made by BVDetector [45] that snippets of vulnerable code frequently include calls to `libc` and other external libraries. We start separately from each of these external library calls in the test program and traverse the DDG in both forward and backward directions. We use the average number of instructions n previously computed from the training data to determine the size of each subgraph to be extracted at test time. Formally, for each external function call instruction c in the function body F, we define the seed set of vulnerable instructions as $S_0 = \{c\}$. Next, by following the DDG in the forward direction, we construct the set S_f of instructions, which are reachable from one of the arguments of c and, by walking in the reverse direction, recover the set S_b of instructions from which one can reach some argument of c. We truncate both S_f and S_b to have no more than $n/2$ assembly instructions each (so that $S_f \cup S_b$ collectively contains no more than n assembly instructions). We submit each of the truncated slices $S_0 \cup S_f \cup S_b$ thus obtained to the classifier. As we show in Section 5.6, this slicing heuristic is very effective in identifying candidate instructions for subsequent classification using the GCN.

In contrast to prior work which also slices the program [26], [27], [45], the main novelty of our approach is in: (*a*) using the library call heuristic only at test time, and (*b*) additionally using data dependencies to precisely target instructions for inclusion in the extracted subgraphs. This allows for accurate production of PDG slices at training time and thus improves the accuracy of our model.

3.4. The Graphical Program Representation

After obtaining the sliced PDGs from the functions being classified, we use graph convolutional networks (GCNs) [22] to train classifiers that can distinguish between vulnerable and bug-free programs. GCNs operate over graphs that can be described as a pair consisting of an adjacency matrix and an assignment of feature vectors to each vertex of the graph. The main challenge that arises in our application is that, depending on the granularity at which the PDG is constructed, each of its nodes will itself have a complex structure. For example, each basic block consists of a sequence of binary instructions, and each instruction in turn consists of an operator applied to several operands. Traditionally, in order to encode these structures into a form suitable for classification and learning, researchers have relied on a range of hand-crafted features. For example, the attributed control flow graphs (ACFGs) of Genius and Gemini involve associating each basic block with a set of manually derived statistics, such as the number of instructions, number of calls, or number of arithmetic instructions that it contains.

In contrast, for vulnerability detection in source code, systems such as Devign [51] rely on an abstract-syntax-tree-like (AST-like) representation augmented with control dependence and data flow edges [49]. Inspired by these techniques, we use the intermediate representation of binary code to construct the abstract syntax tree (AST) for each basic block in the program slice. Each node in this AST corresponds to either an operation (e.g., `INT_ADD`, `COPY`, `LOAD`, etc.) or a var-node. Consequently, there is an edge from the operation performed in the IR statement to the target var-node, and edges from each var-node to IR statements which subsequently use its value. For example, the IR statement `(register, 0x20, 8) INT_ADD (register, 0x28, 8), (register, 0x30, 8)` would have edges from `(register, 0x28, 8)` and `(register, 0x30, 8)` to `INT_ADD`, and from `INT_ADD` to `(register, 0x20, 8)` respectively.

Recall now the granularity difference between data and control flow information that we previously discussed in Section 3.2. Specifically, DDG edges connect operands used in IR statements, while CDG edges connect basic blocks. Our approach to reconciling this discrepancy is to annotate each node in the DDG with some scalar representation of the control dependency information. Motivated by the idea that each basic block in the CDG between the start of the function and the current node can potentially determine whether or not the current node will be executed, we posit that the number of such blocks provides a good scalar encoding of the control dependence information.

We perform a breadth-first search (BFS) over the control dependence graph to recover the level l of each node v, i.e., its distance from the starting block of the function. We finally obtain a graphical representation of the selected slice by embedding this *control dependency level* into each node of the AST, thus resulting in the compound node v_l, as illustrated in Figure 4. Our ablation study in Table 3 indicates that this data is unambiguously helpful in vulnerability detection across all CWE categories.

3.5. Training the Classifier

Feature extraction. For each function slice under consideration (either in training or in test), we construct a graph with a symmetric adjacency matrix, which describes data dependencies from the AST. To extract the features for each node, we consider the set of all nodes across all AST graphs constructed during training. Recall that each of these nodes has a label of the form v_l, where v is an element of the original AST (either an IR opcode or a variable or a constant), and $l \in \{0, 1, 2, \dots\}$ is its control dependency level. We use this global catalog of node labels to develop a one-hot feature encoding for each node in the graph being constructed.

Model setup. Our classifier uses the Kipf GCN architecture [22]. We use the reference implementation, which we have modified to run on TensorFlow 2.8.0. Our model consists of three layers with 128, 128 and 64 dimensions respectively. The learning rate and batch size are set of 0.001 and 64 respectively. In evaluation, depending on the CWE type, we select the number of epochs from the set $\{20, 30, 50\}$ by optimizing over the validation set.

4. Implementation Details

We implemented BinHunter using Ghidra [16] to lift the binary instructions into a language- and architecture-independent intermediate representation called the P-code [34]. Next, we use the Ghidra APIs to recover the control dependence graph (CDG), and data dependence graph (DDG) for the selected functions in the binary. We have shown an example of the combined program dependence graph (PDG) in Figure 3.

Our next step is to apply PDG-slicing. In our evaluation we train the BinHunter classifier using programs from the Juliet dataset. The training programs use preprocessor macros to switch between buggy and bug-free versions of the code, and also include detailed comments regarding the location of the vulnerability. We map the implicated source lines into binary offsets using DWARF debug entries, thereby providing a very accurate seed set V_0 of vulnerable instructions for training.

Comments on the running time. *In terms of asymptotic complexity:* Classical algorithms can construct the PDG in time $O(N^2)$, where N is the number of IR statements in the function. The subgraph slicing heuristic performs a graph traversal from each call node, thus contributing to a total running time of $O(kN)$, where k is the number of call statements within the function body. The final Kipf classifier runs in time linear in the number of edges in the graph being evaluated. *In terms of wall-clock running time:* Obtaining the PDG from Ghidra required an average of 72.61 seconds per function drawn from the real-world dataset, while the subsequent slicing and model evaluation was completed in an average of 3.59 seconds per function.

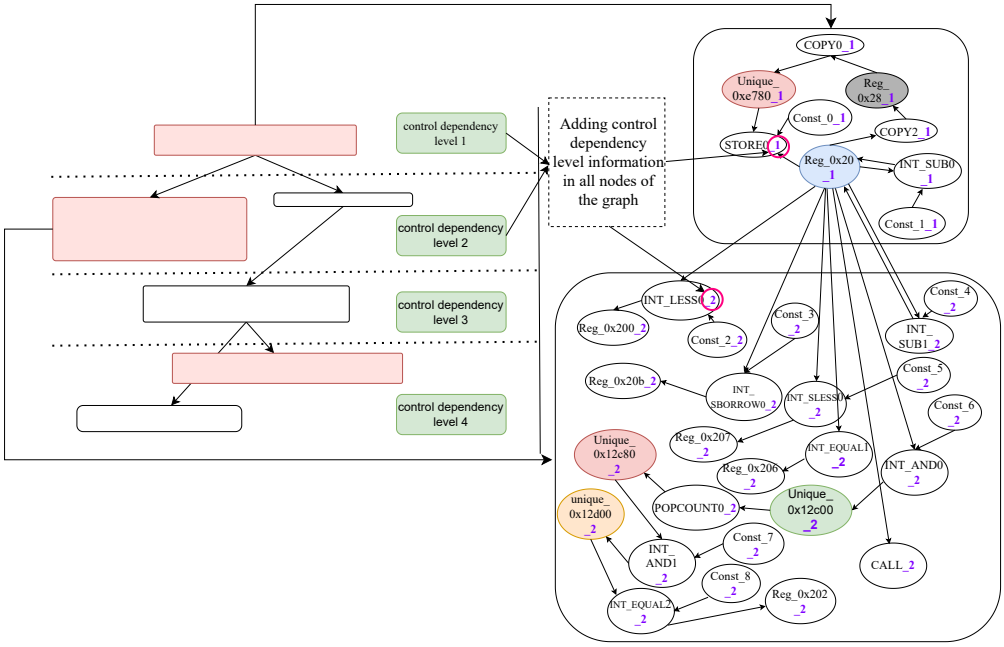

Figure 4: Combining the abstract syntax tree (AST) with the data dependence edges and embedding the control dependence levels to obtain a graphical representation of the function slice.

5. Experimental Results

Our implementation of BinHunter consists of approximately 3,700 lines of Python 3 code and 400 lines of Bash script, and is divided into two principal components. The first component interfaces with Ghidra to extract PDGs and operates using the Ghidrathon [31] Jython-to-Python bridge. We use the `pyelftools` library to parse the DWARF structure, to obtain binary offsets. We performed all experiments on a Linux server with 256 GB RAM and an Intel Xeon E5-1650 CPU with 12 cores. Our evaluation focused on answering the following research questions:

RQ1. How effective is BinHunter at detecting vulnerabilities in both the Juliet test suite and in real-world binary programs?

RQ2. How do various model features, such as slicing, inclusion of data dependency and control dependency features, influence the overall performance of the model?

RQ3. Can BinHunter effectively localize vulnerabilities in functions?

RQ4. Why is slicing based on calls to `libc` effective for detecting vulnerabilities?

Artifact availability. The BinHunter source code and our evaluation dataset may be downloaded from https://github.com/SimaArasteh/binhuntertool/tree/main.

5.1. Benchmarks

We evaluate BinHunter using the Juliet test suite [32] and using a corpus of historically vulnerable Debian packages.

The Juliet test suite. The Juliet Test Suite is a balanced, labeled dataset that consists of vulnerable and patched versions of C, C++, and Java programs [32]. These programs contain examples of 118 different CWEs, including various types of buffer overflows and integer overflows. We focus on samples of C and C++ programs, for which the distribution provides preprocessor macros to switch between the vulnerable and patched versions of the program.

We split the Juliet dataset for each CWE into training, testing, and validation sets at the ratio $70 : 15 : 15$. We subsequently focus on CWEs for which there are at least 100 samples in the test set, and where the patch involves changes to a single function. This yields 17 CWEs and 24,725 binaries. We show the aggregate statistics for the subset of the Juliet test suite, which we use, in Table 2.

Corpus of historical CVEs (Common Vulnerabilities and Exposures). To measure the effectiveness of BinHunter in identifying vulnerabilities in real-world binary programs, we crawled the Debian snapshot archive [38]. We identified packages containing security patches that fix specific CVEs, and manually localized the functions and lines of code being changed in each case. We correlated this information with the NVD vulnerability database [37] to determine the implicated CWE. Through this process, we collected a corpus of 24 CVEs across 15 Debian packages, spanning the time period 2015–2022. We built each of these packages using `debootstrap` (to obtain the historically appropriate version of Debian) and by using the Debian package maintenance tool `quilt` (to apply or withhold the specific patch that fixes the CVE). We thus obtained a balanced dataset of real-world binaries in which specific functions were known

to have or not a given vulnerability. We show the list of CVEs for real-world evaluation in Table 4.

To the best of our knowledge, this is the first such dataset of CVEs, which includes weakness type, affected locations (both at the source and binary levels), and with vulnerable and patched versions of binaries.

TABLE 2: Chosen vulnerability categories from the Juliet test suite. A preprocessor macro provides vulnerable and non-vulnerable versions of each program. We compiled programs with and without this macro, and divided the resulting functions into training, testing, and validation sets, whose respective sizes we present below.

CWE	Training	Testing	Validation
CWE-121	1987	426	426
CWE-122	2195	471	470
CWE-124	659	142	141
CWE-126	499	107	107
CWE-127	659	142	141
CWE-190	2216	476	475
CWE-191	1639	352	351
CWE-369	622	134	133
CWE-400	478	103	102
CWE-401	912	196	196
CWE-415	492	106	106
CWE-416	492	106	106
CWE-476	464	100	100
CWE-590	1176	252	252
CWE-680	478	103	102
CWE-690	536	115	115
CWE-762	1796	386	385

5.2. Baselines

We compare BinHunter to four existing models which aim to detect vulnerabilities in binary programs: Bin2vec [1], Asm2vec [10], Genius [12] and Jtrans [46]. We selected these baselines for the vulnerability discovery task as they each employ comparable methodologies, enabling a direct and meaningful comparison of their respective efficiencies and accuracies in this context.

The first baseline, Bin2vec, is a binary classifier which must be trained separately for each class of vulnerabilities. In contrast, the remaining three baselines, Asm2vec, Genius and Jtrans are primarily binary search tools. Notably, the original evaluations of these tools in the respective publications [10], [12], [46] also include measurements of their effectiveness for vulnerability discovery in binaries. These three baselines apply different techniques: Bin2vec applies GCNs to the inter-procedural CFG extracted from the binary; Asm2vec leverages an NLP model to train vulnerabilities at the granularity of individual functions; Genius has a set of hardcoded features which are manually extracted from the CFG, and passed through a clustering algorithm to identify vulnerable functions; and Jtrans combines a token embedding derived from the assembly instructions with a jump embedding describing control flow information.

We now discuss the evaluation and setup of each system. Note that we use the same partition of functions into training, testing, and validation sets for BinHunter and the baselines.

Bin2vec. We use the same settings (number of GCN layers and training parameters: epochs, learning rate, and batch size) as used in the original Bin2vec evaluation, and separately learn a classifier for each class of vulnerabilities.

Asm2vec. We use the Asm2vec implementation available from [23]. Note that Asm2vec is primarily a bug search engine. We use the system to detect vulnerabilities as follows: we supply each test function to the search engine to search for similar functions in the training repository. Recall that the training functions consist of vulnerable and patched programs from the Juliet test suite. We then classify the test function as being vulnerable if at least half of the top 15 search results were vulnerable. We also note that this setup is analogous to the setup used by the Asm2vec developers in their own evaluation of its effectiveness in vulnerability detection (See Section 5.4 of [10].

Before evaluating the performance of Asm2vec on the Juliet test suite, we use the validation set to identify the best values of the model parameters. This includes the dimensionality of the PV-DM neural network and the learning rate. We evaluate the model parameters by performing a grid search over the sets $\{50, 100, 150, 200\}$ and $\{0.05, 0.025, 0.01\}$ respectively, and choose the setting which performs the best. On the other hand, the limited number of Debian packages precludes the possibility of a validation loop. When running the Asm2vec baseline on this dataset, we therefore use all combinations of parameter values, and report the best classification results ever achieved.

Genius. We use the Genius implementation available from [44]. The system uses the IDA Pro disassembler to extract CFGs from the binary, and uses them to construct a subsequent data structure called an *attributed* CFG (ACFG). We updated the Genius implementation to support the latest version of IDA Pro.

Note that Genius is also a bug search engine similar to Asm2vec, and so we employ a similar technique to use it for vulnerability discovery, i.e., searching through the Juliet repository for functions similar to the sample being classified, and declaring it to be potentially vulnerable if at least half of the top 50 search results are vulnerable. Before measuring the classification performance of Genius, we use the validation set to tune the hyper-parameters (i.e., code-book size, chosen from the range $\{16, 32, 64, 128\}$).

Jtrans. In their original evaluation, the Jtrans developers measured the ability of their system to detect vulnerabilities across versions of a function compiled with different optimization levels. They utilized a corpus of vulnerable binaries compiled with different compilers, compilation flags, and optimization levels. In this corpus, they grouped different versions of the same function, and trained their algorithm to mark functions within the same group as similar, and functions in different groups as dissimilar.

In our setting, on the other hand, we only had one vulnerable and patched version of each binary. While adapting Jtrans to our setting, (for each vulnerability category) we instead

created just two groups of functions, containing vulnerable and patched versions respectively. The smaller number of groups caused some training-time complications with the default Jtrans implementation, which we resolved through minor modifications of their source code.

Finally, as with Asm2vec and Genius: At test time, we requested a ranked list of functions most similar to the function under test, and based our prediction on whether there were more vulnerable or non-vulnerable functions within the top 10 returned results.

5.3. RQ1: Classification Performance

In order to measure their effectiveness in finding vulnerabilities, for each CWE category, we trained all systems on the corresponding training slices of the Juliet test suite. We chose the respective hyper-parameters by measuring the effectiveness of the learned classifier on the validation sets (this includes the number of training epochs for BinHunter, drawn from the range $\{20, 30, 50\}$.

Note that we use DWARF debug information to precisely identify the vulnerable and patched slices during training. We also measured the average number of vulnerable instructions in the training programs to determine the optimal slice size for each CWE, which we report in Table 5.

During testing, we use calls to `libc` as a heuristic to preselect potential locations of vulnerabilities, as discussed in Section 3.3. Depending on the number of these calls in the function being classified, we extract a varying number of slices from each test program. We identify buggy slices by examining their overlap with the known binary offsets of the vulnerability in question. These offsets were determined using comments in the source code of the Juliet programs, and by identifying program locations that were modified by patches applied to the historical vulnerabilities for real-world dataset. We then determined whether the classifier correctly classified the slice, and summarized the results for each function by declaring it to be correctly classified exactly when all of its constituent slices were correctly classified. We repeated this training-testing process 5 times by shuffling the data in each iteration, and measured the average test classification performance. We use two evaluation metrics—true positive rate and false positive rate—to measure the performance of BinHunter on both the Juliet test suite and the historical CVEs. We report these numbers in Tables 3 and 4 respectively.

BinHunter massively outperforms Asm2vec, Genius and Jtrans across all CWE categories in the Juliet dataset. While its performance is closer to Bin2vec, it is nevertheless the best classifier on all but three CWEs (CWE-122, CWE-190, and CWE-590 respectively). Except for CWE-590, BinHunter generally outperforms Bin2vec with a lower false positive rate. On average, BinHunter outperforms the baselines by 6.77%, 26.53%, 24.65% and 41.59% in true positive rates and exhibits lower false positive rates by 19%, 47.64%, 31.47% and 39.82% respectively.

We remark that Bin2vec is primarily based on analyzing the CFG of the program. An investigation of the Juliet programs for CWE-126, CWE-369, CWE-400, and CWE-690 revealed that samples misclassified by Bin2vec were primarily distinguished by their patterns of data flow (as indicated by metadata in the samples and Appendix C of the Juliet manual [32]). This highlights the importance of considering both the control and the data dependencies, as explained in Section 3.4.

We also recall that each vulnerable sample in the Juliet dataset is accompanied by a patched version: because they differ in only a small number of ways, the two versions are likely to be represented by nearby vectors, thereby resulting in their poor classification performance.

When applied to the corpus of historical vulnerabilities, we were able to retrieve the target function using the Ghidra API for 21 of the 24 CVEs.[1] We further classified each function's binary using BinHunter, and using our baselines. We report classification results in Table 4. Note that BinHunter is successful for 17 of the CVEs, while Bin2vec only successfully classifies 2 functions, and Asm2vec, Genius and Jtrans are uniformly unsuccessful. The data presented in Table 4 indicates that BinHunter does not successfully identify vulnerable slices associated with CVE-2019-12109. Additionally, for CVE-2021-37529, CVE-2022-31291, and CVE-2017-7395, it incorrectly labels some or all non-vulnerable slices as vulnerable. The vulnerable functions of CVEs in Table 4 are quite large. The reliance of Bin2vec, Asm2vec, Genius and Jtrans on function representations affects their performance on real-world binaries, while BinHunter relies on a finer granularity level, which improves its accuracy.

5.4. RQ2: Ablation Study

Next, we conduct three experiments to study the impact of different features on the ultimate performance of BinHunter. First, we remove control dependency information from the model, and evaluate an alternative model that only operates on the sliced data dependency graph. We call this model BH $\setminus$ $\{$CDG$\}$. We then inspect the impact of slicing on the ultimate accuracy of vulnerability detection using the model BH $\setminus$ $\{$Slicing$\}$, which learns and performs classification over the entire functions instead of function slices. Finally, we study the impact of the data dependency edges on classification performance. Recall that nodes in the overall BinHunter graph are connected through data dependency edges, and information about control dependencies is merely embedded into each node in the graph. This makes setup of this final ablation study somewhat challenging. Inspired by the graph representation in Bin2vec, we construct the AST of each basic block, augmented with source and sink nodes at its start and end. We add edges between these start and end vertices according to the control dependency graph. We employ a slicing technique similar to the main model, and denote this derived model as BH $\setminus$ $\{$DDG$\}$. The accuracy of these models may be found in the corresponding columns of Tables 3 and 4.

1. The Ghidra API fails to retrieve the vulnerable function for three CVEs: CVE-2016-5152, CVE-2016-7163, and CVE-2018-5785.

TABLE 3: Evaluation of BinHunter and the baseline systems on the Juliet dataset. Note that TPR and FPR shows true positive and false positive rates respectively.

CWE-ID	# Slices of BinHunter	BinHunter		BH \ {CDG}		BH \ {DDG}		BH \ {Slicing}		Bin2vec		Asm2vec		Genius		jTrans	
		TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR
CWE-121	1329	0.83	**0.07**	0.81	0.29	0.75	0.30	**0.85**	0.15	0.82	0.49	0.52	0.43	0.67	0.59	0.38	0.70
CWE-122	2844	0.81	**0.09**	0.72	0.28	0.74	0.32	0.80	0.08	**0.85**	0.29	0.54	0.61	0.50	0.21	0.40	0.59
CWE-124	738	**0.84**	**0.04**	0.83	0.08	0.83	0.13	0.82	0.10	0.66	0.30	0.78	0.22	0.55	0.61	0.27	0.40
CWE-126	699	**0.86**	**0.07**	0.83	0.36	0.77	0.46	0.80	0.12	0.63	0.23	0.69	0.37	0.42	0.43	0.37	0.55
CWE-127	838	**0.83**	**0.06**	0.81	0.20	0.79	0.45	0.81	0.17	0.78	0.21	0.72	0.56	0.75	0.34	0.31	0.47
CWE-190	2851	0.81	**0.09**	0.83	0.17	0.84	0.23	0.83	0.12	**0.85**	0.25	0.57	0.67	0.52	0.47	0.45	0.65
CWE-191	2190	**0.88**	**0.10**	0.80	0.12	0.84	0.2	0.81	0.10	0.81	0.23	0.63	0.42	0.49	0.32	0.40	0.77
CWE-369	873	**0.88**	**0.08**	0.82	0.15	0.81	0.26	0.87	0.27	0.78	0.31	0.58	0.57	0.67	0.31	0.52	0.65
CWE-400	864	**0.91**	**0.05**	0.80	0.18	0.79	0.42	0.89	0.15	0.76	0.31	0.42	0.73	0.70	0.13	0.31	0.44
CWE-401	661	**0.80**	**0.06**	0.82	0.16	0.77	0.16	0.81	0.17	0.73	0.21	0.58	0.63	0.65	0.23	0.24	0.58
CWE-415	356	**0.82**	**0.06**	0.81	0.06	0.79	0.08	0.78	0.20	0.80	0.1	0.62	0.76	0.61	0.55	0.56	0.3
CWE-416	356	**0.82**	**0.05**	0.72	0.17	0.70	0.23	0.82	0.26	0.76	0.24	0.33	0.80	0.59	0.21	0.71	0.46
CWE-476	156	**0.95**	**0.12**	0.86	0.25	0.80	0.36	0.83	0.14	0.80	0.33	0.55	0.46	0.53	0.42	0.72	0.13
CWE-590	866	0.79	0.04	0.82	**0.01**	0.83	0.19	**0.86**	0.10	0.80	**0.01**	0.68	0.51	0.61	0.24	0.57	0.28
CWE-680	441	**0.88**	**0.03**	0.87	0.05	0.85	0.17	0.83	0.15	0.82	0.18	0.52	0.43	0.68	0.48	0.38	0.17
CWE-690	328	**0.83**	**0.05**	0.80	0.02	0.80	0.12	0.82	0.09	0.79	0.29	0.47	0.51	0.59	0.56	0.43	0.44
CWE-762	1220	**0.80**	**0.03**	0.77	0.08	0.67	0.03	0.80	0.23	0.75	0.34	0.63	0.51	0.62	0.34	0.25	0.28

TABLE 4: Classification performance on the corpus of historical vulnerabilities over 5 random runs. We also report the affected package and implicated CWE. # Slices indicates the total number of slices created by BinHunter. Note that Bin2vec and Asm2vec fail to execute on CVE-2021-32280, CVE-2018-20190 and CVE-2021-32280 because of a crash in angr. Bin2vec also crashes when running on CVE-2019-14818.

CVE	Debian Package	CWE-ID	BinHunter			BH \ {CDG}		BH \ {DDG}		BH \ {Slicing}		Bin2vec		Asm2vec		Genius		jTrans	
			# Slices	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR
CVE-2015-8871	openjpeg2	CWE-416	11	1	0	0.4	0.4	0	0	0.2	0	0	0	0	0	0	0	0	0
CVE-2019-11471	libheif	CWE-416	24	1	0	0.6	0	0	0	0	0	-	-	-	-	0	0	0	1
CVE-2020-1983	libslirp	CWE-416	22	1	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0
CVE-2020-27841	openjpeg2	CWE-122	13	1	0	0.4	0	0	0	0	0	0.6	0	0	0	0	0	0	0
CVE-2022-1253	libde265	CWE-122	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CVE-2017-7458	ntopng	CWE-476	5	1	0	0.6	0	0	0	1	1	0	0	0	0	0	1	0	0
CVE-2018-20349	r-cran-igraph	CWE-476	19	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
CVE-2018-13441	nagios4	CWE-476	3	1	0	0.6	0	0	0	0	0	0	0	0	0	0	0	0	0
CVE-2018-20190	libsass	CWE-476	4	1	0	0	0	0	0	0	0	-	-	-	-	0	0	0	0
CVE-2019-18388	virglrenderer	CWE-476	17	1	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
CVE-2019-12110	miniupnpd	CWE-476	22	1	0	0.4	0	0	0	0	0	0	0	0	0	0	0	0	0
CVE-2019-12109	miniupnpd	CWE-476	7	0	0	0	0	0	0	0	0.2	0	0	0	0	0	0	0	0
CVE-2019-12108	miniupnpd	CWE-476	4	1	0	0	0	0	0	0	0.2	0	0.2	0	0	0	0	0	1
CVE-2021-32280	fig2dev	CWE-476	9	1	0	0	0	0	0	0	0	-	-	-	-	0	1	0	0
CVE-2020-8003	virglrenderer	CWE-415	16	1	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0
CVE-2021-37529	fig2dev	CWE-415	5	0	0.33	0	0	0	0	1	1	1	0	0	0	0	1	0	0
CVE-2022-31291	dlt-daemon	CWE-415	9	0	0.4	0	0	0	0	0	0	1	0	0	0	0	1	0	0
CVE-2019-14818	dpdk	CWE-401	14	1	0	0	0.5	0	0	0	0	-	-	0	0	0	0	0	0
CVE-2017-7395	tigervnc	CWE-190	8	0	1	0.66	0	1	1	0	0	0	0	0	0	0	0	0	0
CVE-2020-10722	dpdk	CWE-190	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
CVE-2022-48468	libsignal-protocol-c	CWE-190	20	1	0	0.2	0	0	0	0	0	0	0	0	0	0	0	0	0

The approach without slicing – BH \ {Slicing} – has 97.8% of the TPR and 239% of the FPR of the final model, when applied to the Juliet test suite. On the other hand, its average TPR over the Debian packages drops to only 18.8% of the final model. This observation highlights the importance of slicing the PDG when detecting vulnerabilities in large real-world functions.

The importance of slicing is further highlighted in the case of BH\{CDG}. Observe that despite performing slightly worse than BH \ {Slicing} over the Juliet test suite, it has much greater accuracy in the case of Debian vulnerabilities, with 52% higher TPR and 62.5% lower TPR than the model which classified entire functions, BH \ {Slicing}.

Finally, the model BH\{DDG} performs poorly on both the Juliet test suite and Debian packages, highlighting the critical role of data dependency information in discovering vulnerabilities in these environments.

5.5. RQ3: Effectiveness in Localization

Next, we attempted to measure how effective BinHunter is in flagging small slices of the binary as being potentially vulnerable. For each program, we considered the set of potentially vulnerable instructions returned by BinHunter and compared them to ground truth extracted from the patch and code comments in the dataset. We measured the fractional overlap between these sets, defined as the values $|V \cap S|/|V|$ and $|V \cap S|/|S|$, where V and S are the ground truth set of vulnerable instructions and the program slice flagged by BinHunter respectively. We present this data in Table 5 for Juliet test suite and Table 6 for the Debian packages.

Observe that the average values for both quantities exceed 80% and 70% respectively for the Juliet test suite, indicating that the slices produced are highly correlated with the ground truth. Also observe that the slices extracted by BinHunter contain 30 instructions on average, indicating that they are potentially small enough for subsequent manual analysis.

Over the Debian packages, these measures of localization effectiveness exceed 80% and 65% respectively. Observe

TABLE 5: Average number of vulnerable instructions in the training data, and the measured overlap (in %) between flagged instructions and the ground truth during testing on the Juliet test suite dataset. Here V and S denote the set of vulnerable instructions (i.e., the ground truth) and the set of instructions returned by BinHunter respectively.

CWE-ID	Avg Instructions	$\lvert V \cap S\rvert/\lvert V\rvert$	$\lvert V \cap S\rvert/\lvert S\rvert$
CWE-121	26	89.49	80.71
CWE-122	26	90.21	76.25
CWE-124	33	90.41	91.17
CWE-126	34	95.23	62.04
CWE-127	30	92.52	63.62
CWE-190	27	80.56	82.31
CWE-191	27	92.32	85.73
CWE-369	29	90.41	80.23
CWE-400	52	88.80	87.86
CWE-401	52	72.44	62.83
CWE-415	15	95.90	90.50
CWE-416	14	85.30	80.70
CWE-476	29	89.33	81.83
CWE-590	17	87.27	92.60
CWE-680	53	90.50	60.51
CWE-690	17	93.39	65.92
CWE-762	16	93.95	71.76

TABLE 6: Measurement of localization effectiveness over the Debian packages.

CVE-ID	$\lvert V \cap S\rvert/\lvert V\rvert$	$\lvert V \cap S\rvert/\lvert S\rvert$
CVE-2015-8871	71.42	53.27
CVE-2019-11471	70.29	65.14
CVE-2020-1983	81	63
CVE-2020-27841	87	57.12
CVE-2022-1253	88	61.17
CVE-2017-7458	75	70.20
CVE-2018-20349	82	57.20
CVE-2018-13441	85.21	71.2
CVE-2018-20190	89.41	70.32
CVE-2019-18388	70	54
CVE-2019-12110	76.12	63.24
CVE-2019-12109	78	63.5
CVE-2019-12108	87.25	57.5
CVE-2021-32280	85.29	64.07
CVE-2020-8003	81.32	71.21
CVE-2021-37529	75	55.55
CVE-2022-31291	85.71	66.65
CVE-2019-14818	83.33	61.2
CVE-2017-7395	75.07	57.14
CVE-2020-10722	78.56	74.30
CVE-2022-48468	84.44	64.27

that although these numbers are very large, the value of $|V \cap S|/|S|$ is smaller than in the Juliet dataset. We speculate that this is because of the complexity of data dependencies in real world programs, which forces the algorithm to create large slices. Nevertheless, these slices remain sufficiently small to reduce the effort required for subsequent manual analysis.

5.6. RQ4: Effectiveness of the Slicing Heuristic

As described in Section 3.3, we use calls to external functions as the locations from which to extract program slices for subsequent classification by BinHunter. In order to assess the effectiveness of this heuristic, we performed an analysis of the training and testing fragments of the Juliet dataset, and counted the functions whose vulnerable slices contain such external calls. We present this data in Figure 5. The first column shows the measurement over the training set while the second column presents measurements over the testing set.

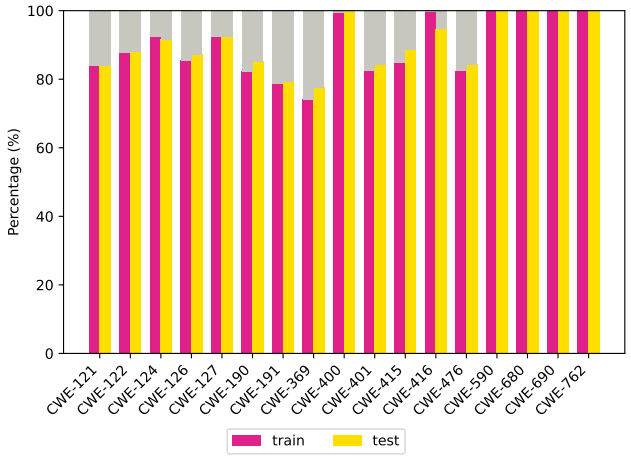

Figure 5: Fraction of vulnerable slices in the Juliet dataset that contain calls to external functions.

Note that in the specific Juliet programs, these calls usually target functions in `libc`. Observe that the heuristic consistently captures a large fraction of the vulnerable slices: the heuristic always captures at least 80% of vulnerable slices, and for 6 of the 17 CWEs in question, the heuristic manages to capture *all* vulnerable slices. The CWEs for which the method has the smallest recall, i.e., CWE-121, CWE-190, CWE-191, and CWE-369, correspond to stack-based buffer overflows, integer overflow and underflows, and divide-by-zero respectively. These vulnerabilities might conceivably be exploited without calls to external libraries, thus potentially explaining the lower recall of our slicing heuristic in these cases.

6. Challenges and Limitations

We now discuss some limitations of our classification technique, which we plan to address as part of future research.

First, we restricted our focus to intra-procedural vulnerabilities. This was primarily due to challenges in recovering long-range data dependencies. On the other hand, binary analysis backends such as angr [43] provide some support for also obtaining inter-procedural data dependencies. We plan to investigate the feasibility of using this information for vulnerability detection as part of future work.

Next, BinHunter would be unable to distinguish between the vulnerable and patched versions of the program if they only differ in the values of constants, such as lengths of arrays. Note that this is not just a limitation of our present

technique, but is also a challenge for most other deep learning based vulnerability detectors. We plan to investigate richer encodings, which might also provide information such as the ordering of constants to help address this class of limitations.

7. Conclusion

In this paper, we studied the problem of using graph convolutional networks for vulnerability detection in binaries. Our technique, BinHunter, constructs subgraphs of the PDG to identify relevant portions of the function being tested, and learns a novel graph representation using the sliced PDG. In addition to flagging potential vulnerabilities, our technique is therefore also able to localize the warning to small fragments of the function being examined. Experiments with programs from the Juliet test suite and a dataset of vulnerable Debian packages indicate that our technique outperforms existing binary vulnerability detectors.

References

[1] Shushan Arakelyan, Sima Arasteh, Christophe Hauser, Erik Kline, and Aram Galstyan. Bin2vec: learning representations of binary executable programs for security tasks. *Cybersecurity*, 4(1):1–14, 2021.

[2] Amy Aumpansub and Zhen Huang. Learning-based vulnerability detection in binary code. In *2022 14th International Conference on Machine Learning and Computing (ICMLC)*, pages 266–271, 2022.

[3] Hesam Azadjou, Ali Marjaninejad, and Francisco Valero-Cuevas. Play it by ear: A perceptual algorithm for autonomous melodious piano playing with a bio-inspired robotic hand. *bioRxiv*, pages 2024–06, 2024.

[4] Bita Azarijoo, Mostafa Salehi, and Shaghayegh Najari. A meta path-based approach for rumor detection on social media. *arXiv preprint arXiv:2301.04341*, 2023.

[5] Sara Baradaran, Mahdi Heidari, Ali Kamali, and Maryam Mouzarani. A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes. *International Journal of Information Security*, pages 1–14, 2023.

[6] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a" siamese" time delay neural network. *Advances in neural information processing systems*, 6, 1993.

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

[8] Kenneth Ward Church. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017.

[9] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711. PMLR, 2016.

[10] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[11] Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. discovre: Efficient cross-architecture identification of bugs in binary code. In *Ndss*, volume 52, pages 58–79, 2016.

[12] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 480–491, 2016.

[13] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[15] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, Heyuan Shi, and Jiaguang Sun. Vulseeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 803–808, 2018.

[16] Ghidra. Ghidra reference. https://ghidra-sre.org/, 2018.

[17] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 85–96, 2016.

[18] Arash Hajisafi, Haowen Lin, Yao-Yi Chiang, and Cyrus Shahabi. Dynamic gnns for precise seizure detection and classification from eeg data. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 207–220. Springer, 2024.

[19] Arash Hajisafi, Haowen Lin, Sina Shaham, Haoji Hu, Maria Despoina Siampou, Yao-Yi Chiang, and Cyrus Shahabi. Learning dynamic graphs from all contextual information for accurate point-of-interest visit forecasting. In *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems*, pages 1–12, 2023.

[20] 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*, pages 702–715, 2021.

[21] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[22] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[23] Lancer. asm2vec: Learning program vector representations for binary code similarity analysis, 2018. https://github.com/Lancern/asm2vec.

[24] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[25] Young Jun Lee, Sang-Hoon Choi, Chulwoo Kim, Seung-Ho Lim, and Ki-Woong Park. Learning binary code with deep learning to detect software weakness. In *KSII the 9th international conference on internet (ICONI) 2017 symposium*, 2017.

[26] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

[27] 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. *arXiv preprint arXiv:1801.01681*, 2018.

[28] Shigang Liu, Mahdi Dibaei, Yonghang Tai, Chao Chen, Jun Zhang, and Yang Xiang. Cyber vulnerability intelligence for internet of things binary. *IEEE Transactions on Industrial Informatics*, 16(3):2154–2163, 2019.

[29] Zian Liu, Lei Pan, Chao Chen, Ejaz Ahmed, Shigang Liu, Jun Zhang, and Dongxi Liu. Vulmatch: Binary-level vulnerability detection through signature. *arXiv preprint arXiv:2308.00288*, 2023.

[30] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. In *NDSS*, 2023.

[31] Mandiant. Ghidrathon GitHub Repository. https://github.com/mandiant/Ghidrathon, 2023.

[32] FGG Meade. Juliet test suite v1. 2 for c/c++ user guide. *no. December*, 2012.

[33] Maryam Mouzarani, Ali Kamali, Sara Baradaran, and Mahdi Heidari. A unit-based symbolic execution method for detecting heap overflow vulnerability in executable codes. In *International Conference on Tests and Proofs*, pages 89–105. Springer, 2022.

[34] Nico Naus, Freek Verbeek, Dale Walker, and Binoy Ravindran. A formal semantics for p-code. In *Verified Software. Theories, Tools and Experiments.*, pages 111–128, Cham, 2023. Springer.

[35] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14, 2001.

[36] Hong Nguyen, Arash Hajisafi, Alireza Abdoli, Seon Ho Kim, and Cyrus Shahabi. An evaluation of time-series anomaly detection in computer networks. In *2023 International Conference on Information Networking (ICOIN)*, pages 104–109. IEEE, 2023.

[37] National Institute of Standards and Technology (NIST). National vulnerability database (nvd), 2023. https://nvd.nist.gov/.

[38] Debian Project. Debian package tracker – snapshot service, 2023. https://snapshot.debian.org/.

[39] Parsa Razmara, Tina Khezresmaeilzadeh, and B Keith Jenkins. Fever detection with infrared thermography: Enhancing accuracy through machine learning techniques. *arXiv preprint arXiv:2407.15302*, 2024.

[40] Andreas Schaad and Dominik Binder. Deep-learning-based vulnerability detection in binary executables. In *International Symposium on Foundations and Practice of Security*, pages 453–460. Springer, 2022.

[41] Sina Shaham, Arash Hajisafi, Minh K Quan, Dinh C Nguyen, Bhaskar Krishnamachari, Charith Peris, Gabriel Ghinita, Cyrus Shahabi, and Pubudu N Pathirana. Holistic survey of privacy and fairness in machine learning. *arXiv preprint arXiv:2307.15838*, 2023.

[42] Noam Shalev and Nimrod Partush. Binary similarity detection using machine learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, pages 42–47, 2018.

[43] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[44] Yunlong Song. Genius: Generic neural shape analysis, 2019. https://github.com/Yunlongs/Genius.

[45] Junfeng Tian, Wenjing Xing, and Zhen Li. Bvdetector: A program slice-based binary code vulnerability intelligent detection system. *Information and Software Technology*, 123:106289, 2020.

[46] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, 2022.

[47] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 363–376, 2017.

[48] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 376–387, 2020.

[49] 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. IEEE, 2014.

[50] Michał Zalewski. American fuzzy lop - whitepaper, 2016.

[51] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[52] Xiaogang Zhu, Sheng Wen, Alireza Jolfaei, Mohammad Sayad Haghighi, Seyit Camtepe, and Yang Xiang. Vulnerability detection in siot applications: A fuzzing method on their binaries. *IEEE Transactions on Network Science and Engineering*, 9(3):970–979, 2020.