ISGraphVD: Precise Vulnerability Detection for IoT Supply Chains Based on Identifier Sensitive Graph

Yingli Zhang[1], Xin Liu[1*], Ziang Liu[1], Song Li[2*], Nan Li[1], Weina Niu[3], Rui Zhou[1], Qingguo Zhou[1*]
[1] Lanzhou University, {120220909270, bird, liuza20, 220220941890, zr, zhouqg}@lzu.edu.cn
[2] Zhejiang University, songl@zju.edu.cn
[3] University of Electronic Science and Technology of China, vinusniu@uestc.edu.cn

***Abstract*—Open-source software (OSS) is widely reused in Internet of Things (IoT) devices, leading to widespread N-Day vulnerabilities when outdated components remain unpatched. Existing methods typically encode features of different Common Vulnerabilities and Exposures (CVEs) within a shared representation space. However, the model's limited capacity, combined with the new vulnerability features, can disrupt previously learned patterns. Minimal code modifications in tiny-patch vulnerabilities are often overshadowed by variations introduced by different compilation settings, making it more difficult to distinguish vulnerable functions from their patched counterparts. This paper introduces ISGraphVD, a novel graph-based and function-level vulnerability detection approach that supports cross-compilation settings and enhances detection accuracy. By modeling each CVE independently through a one-model-per-CVE strategy, ISGraphVD reduces feature interference and improves detection accuracy across diverse CVEs. To better detect tiny-patch vulnerability, we propose ISGraph, a fine-grained graph representation that models variable dependencies within and across basic blocks by integrating control flow analysis. Then, ISGraphVD utilizes a Graph Matching Network (GMN) with a cross-graph attention mechanism to identify critical vulnerability patterns. Experiments on IoT OSS projects show that ISGraphVD outperforms state-of-the-art methods, achieving a 6.3 percentage-point (pp) accuracy improvement over the strongest baseline, and real-world tests further validate its effectiveness in IoT supply chains.**

***Index Terms*—Binary Code, N-Day Vulnerabilities, Data Dependence, Open-Source Software, Vulnerability Detection, Graph Matching Network**

I. Introduction

Open-source software (OSS) is widely used in Internet of Things (IoT) development due to its availability and flexibility, accelerating time-to-market [1], which leads to widespread code reuse in IoT devices. Maintenance of OSS usually patches disclosed vulnerabilities through version updates. However, IoT devices face constraints in computational and storage resources, limiting the implementation of complex security measures and monitoring. As a result, devices may run outdated software with N-Day vulnerabilities for extended periods, creating significant attack windows for hackers [2], [3]. Thus, effectively detecting widespread N-Day vulnerabilities caused by code reuse in IoT devices is crucial.

Due to varying usage requirements, IoT firmware is developed by different architectures, compilers, compiler versions, and optimization levels, making the detection of N-Day vulnerabilities in IoT devices more challenging. Binary Code Similarity Detection (BCSD) enables determining whether two binary code fragments (e.g., functions) are semantically similar or homologous. Therefore, it can be used to identify N-day vulnerabilities in IoT firmware by detecting the presence of functions that are semantically similar to vulnerable functions from OSS. Recently, deep learning techniques have gained popularity in BCSD for their high accuracy and ability to automatically learn complex features [4]. These approaches extract high-level representations from binary files, including assembly instructions [5], [6], intermediate representations (IR) [7], [8], structure [9]–[11], and data flow [12], etc. These representations are subsequently encoded into numerical vectors and processed by models to generate function-level features. However, to identify similar functions across different compilation settings, current approaches typically have two limitations.

C1: Binaries are compiled under various compilation environments (such as different architectures, compilers, compiler versions, and optimization levels), resulting in binary functions with the same semantics but different structures. Many approaches have supported vulnerability detection across different compilation environments, but these methods [5], [7], [13] typically employ a shared learning process where multiple vulnerability characteristics are mixed within the same model, rather than being independently modeled for each vulnerability. This lack of isolation often limits the model's ability to generalize to new vulnerabilities, as incorporating additional vulnerability features may interfere with previously learned patterns, potentially reducing detection accuracy for known CVEs.

C2: Effectively distinguishing between vulnerable functions and their patched counterparts is crucial for reducing the false positive rate (FPR) in vulnerability detection. Although some studies [14]–[16] incorporate patch information to differentiate patched and vulnerable functions, their detection accuracy drops significantly for vulnerabilities with minimal modifications—sometimes involving only a single statement or even just a variable. We refer to these as **Tiny-Patch** vulnerabilities. This decline occurs because such minor patches are often overshadowed by compilation-induced variations, making it difficult to distinguish patched functions from their vulnerable

* Xin Liu, Song Li and Qingguo Zhou are corresponding authors.

counterparts.

To overcome these limitations, we introduce ISGraphVD (**I**dentifier **S**ensitive **G**raph **V**ulnerability **D**etection), a novel approach for binary vulnerability detection that robustly supports cross-compilation environments and improves detection accuracy. To tackle the first limitation, ISGraphVD adopts a **one-model-per-CVE** strategy, ensuring that the features of different vulnerabilities are learned independently in separate models rather than being mixed within a shared representation space. This isolation mitigates the interference between different vulnerability patterns and enhances the model's generalization capability, as discussed in C1. Each model captures the distinct signature of a specific CVE, enabling ISGraphVD to build a signature-based model database for known vulnerabilities. When analyzing a binary, ISGraphVD compares all functions against the independently trained vulnerability models in the database. Furthermore, when a new vulnerability emerges, a dedicated model is trained and seamlessly integrated into the database without affecting the detection of existing CVEs.

To address the second limitation, we propose a fine-grained graph representation called the Identifier Sensitive Graph (ISGraph). We introduce RT (Read-Tracking) dependencies, which explicitly capture all read operations of a variable, ensuring a more comprehensive representation of its usage. Building on this, we further incorporate control flow structural analysis to model RT, RAW (Read-After-Write), WAR (Write-After-Read), and WAW (Write-After-Write) dependencies arising from variable read and write operations both within and across basic blocks. These dependencies are then explicitly represented as edges between variable nodes in the CPG, enabling a multi-level semantic representation across blocks, statements, expressions, and variables. Additionally, ISGraphVD employs a Graph Matching Network (GMN) [17] with a cross-graph attention mechanism to extract rich semantic information from ISGraph and measure similarity based on the distance between paired function embeddings. Since vulnerabilities typically affect only a small portion of a function, the critical subgraphs or paths in its graph representation carry greater significance than other regions. The attention mechanism in GMN assigns higher weights to these key subgraphs, allowing the model to focus on the most relevant vulnerability-related structures and effectively tackle the challenge C2.

We implement a prototype of ISGraphVD and evaluate it through experiments. We release the ISGraphVD program[1] to facilitate follow-up research. Results show that ISGraphVD excels in vulnerability detection and supports cross-compilation environments. Its superior performance over VulSeeker [7], Gemini [18], and FIT [16] can be attributed to the adoption of a one-model-per-CVE strategy, which avoids feature interference and enhances the extraction of vulnerability-specific code semantics. Furthermore, we use ISGraphVD to conduct real-world experiments, validating its practical applicability in detecting vulnerabilities across diverse software projects. Additionally, we perform time efficiency evaluations to assess the computational cost of our approach. While the one-model-per-CVE strategy increases the initial training cost, it proves to be a one-time investment, as once trained, the model can efficiently detect vulnerabilities without requiring frequent retraining. Experimental results demonstrate that this trade-off is acceptable.

Our contributions are summarized as follows:

- We propose ISGraphVD, a graph-based vulnerability detection approach designed to be robust across different architectures, compilers, compiler versions, and optimization settings. This approach ensures high detection accuracy while minimizing false alarms.
- ISGraphVD introduces an innovative one-model-per-CVE strategy, allowing each model to specialize in capturing the unique characteristics of a specific CVE while preventing interference between different vulnerability patterns.
- We are the first to highlight the impact of tiny-patch vulnerabilities on detection accuracy. To address this, we propose ISGraph, a fine-grained graph representation method that models data dependencies within and across basic blocks caused by read and write operations under different control structures. By introducing RT dependencies, ISGraph overcome the limitation of traditional CPGs, where RAW dependencies capture only the first read after a variable is written, failing to track multiple usages of the same value across different locations.
- We implements a prototype of ISGraphVD, evaluated on widely used open-source projects in IoT devices. Experimental results demonstrate that ISGraphVD achieves superior performance in terms of AUC and detection accuracy compared to existing methods. Furthermore, real-world testing validates its effectiveness in accurately identifying vulnerabilities across both pre- and post-patch versions.

II. Overview

In this section, we explain why we chose pseudo-code to represent binary files and discuss the main problems addressed in this paper.

A. Unified Binary Representation

This paper adopts pseudo-code as a unified representation for binaries. Pseudo-code, typically generated through decompilation tools such as IDA Pro [19] and Ghidra [20], offers a human-readable approximation of the source code. The decompilation process abstracts away hardware-specific details, enabling cross-architecture analysis by translating binaries from various architectures into a common C-like language. Decompilation can reconstruct control structures, such as different types of loops and branching structures. Compared to assembly code and IR, it can recover higher-level semantic information, making pseudo-code an ideal choice for representing binary files in this study [21], [22].

[1]https://github.com/SQJDXLL/ISGraphVD

B. Motivation

This section explains the common problems in the current work and our motivation for constructing a novel graph representation for binary functions.

Fig. 1(A) depicts the source code for a motivating example of a tiny-patch vulnerability, CVE-2015-8899, a denial-of-service vulnerability in Dnsmasq version 2.75. The vulnerability was patched by adding a null pointer check "&& addr", preventing access to its members when variable addr is null. The patch code accounts for only 0.64% of the entire function. Fig. 1(B) and (C) compare the partial pseudo-code of the vulnerable and patched versions, decompiled from binaries compiled with different compilation settings, including variations in architecture (ARM and AARCH64), compiler (GCC and Clang), and optimization levels (O0 and Os). In Fig. 1(B), the variable corresponding to addr is cp. When cp is a null pointer, the condition "cp == 0LL" evaluates to true. Consequently, in the if statement on line 13, the value of "!v13" becomes false, preventing the execution of lines 14–21 and thereby avoiding access to the members of cp. In Fig. 1(C), the variable corresponding to addr is a2. When a2 is a null pointer, the condition "!a2" evaluates to true, making the if statement on line 5 evaluate to true. As a result, execution jumps to LABEL28, bypassing lines 7–16 and preventing access to a2's members. Despite structural differences in the pseudo-code between B and C, their core semantics remain consistent. However, when comparing the pre- and post-patch versions in C, the fix consists of merely adding a "!a2" check, which is far smaller than the differences caused by varying compilation settings in B and C. This observation highlights a key challenge in detecting tiny-patch vulnerabilities: while the actual patch involves minimal code modifications, the differences introduced by different compilation settings can overshadow these subtle changes, making it harder to identify the actual fix-related modifications. Therefore, effectively extracting and emphasizing vulnerability-related features from the compilation-induced redundant information is essential for improving detection accuracy.

Tiny-patch vulnerabilities often involve minimal code modifications, sometimes just a single statement or variable. Enhancing variable-level sensitivity is crucial for capturing key vulnerability patterns and detecting subtle patch-related changes. By tracking how a variable's value is read and modified, we can extract more stable semant ic features while reducing reliance on structural code changes. Code Property Graphs (CPGs) [23] are language-agnostic program representations that merge abstract syntax trees, control-flow graphs, and data-flow graphs into a unified graph structure, and are widely used for static vulnerability discovery in binary analysis. Traditional CPGs employ RAW, WAR, and WAW dependencies to track variable interactions. However, RAW dependencies only capture the first read of a variable after it has been written, failing to track multiple usages of the same value across different locations—an essential factor in understanding its full impact. To address this limitation, we introduce Read-Tracking (RT) dependencies, which track all read operations of a variable. This allows for a more comprehensive data flow analysis by capturing how a variable's value propagates across multiple usage points, rather than being limited to its first read. Consequently, we model variable value propagation using RT, RAW, WAR, and WAW dependencies. Beyond intra-basic-block interactions, our approach incorporates inter-basic-block dependencies through control flow analysis. This enables the capture of long-range data flow patterns, providing a more comprehensive representation of how a vulnerability fix impacts a function's execution logic rather than just localized statement-level modifications. Furthermore, while CPGs encode data dependencies through edges between statement or expression nodes, they do not explicitly specify which variable contributes to these dependencies. To overcome this, we explicitly represent variable dependencies as edges between variable nodes within the CPG, providing a more comprehensive representation of data flow across multiple granularities, including blocks, statements, expressions, and variables.

Additionally, to precisely capture vulnerability-related features from the fine-grained graph, we abandon rule-based slicing methods and instead adopt GMN enhanced with a cross-graph attention mechanism. This enables the model to effectively identify differences between vulnerable and patched code while also recognizing similarities between vulnerable code compiled with different compilation settings. As a result, ISGraphVD learns more robust vulnerability feature representations, thereby improving detection accuracy across various compilation settings.

III. Proposed Approach

A. Overall Framework

The overall workflow of ISGraphVD is shown in Fig. 2. ISGraphVD is a function-level vulnerability detection approach with a "multi-model" design. Unlike most existing methods that use a single model for all vulnerabilities, we train a separate model for each CVE. Each model functions as a dedicated checker, and when a zero-day vulnerability is reported, a new checker is trained specifically for it and added to the existing checker group. The process of training a detection model proceeds as follows: First, the pre- and post-patch versions of the project source code are used to construct the dataset (see Section IV.B for details). Next, the binary files in the dataset are decompiled into high-level representations. ISGraphs are then extracted from these decompiled codes. Subsequently, the ISGraphs are organized into similar and dissimilar pairs and fed into GMN, which predicts whether the input functions are patched or unpatched.

Assumption We assume that the source code of both the pre-patch and post-patch versions can be accessed via version control systems (e.g., Git). Our system mainly targets vulnerabilities in the C language, which has more severe vulnerabilities than other programming languages. Over the past decade, 52.13% of reported vulnerabilities in open-source software have been found in C/C++ [24]. Our design approach

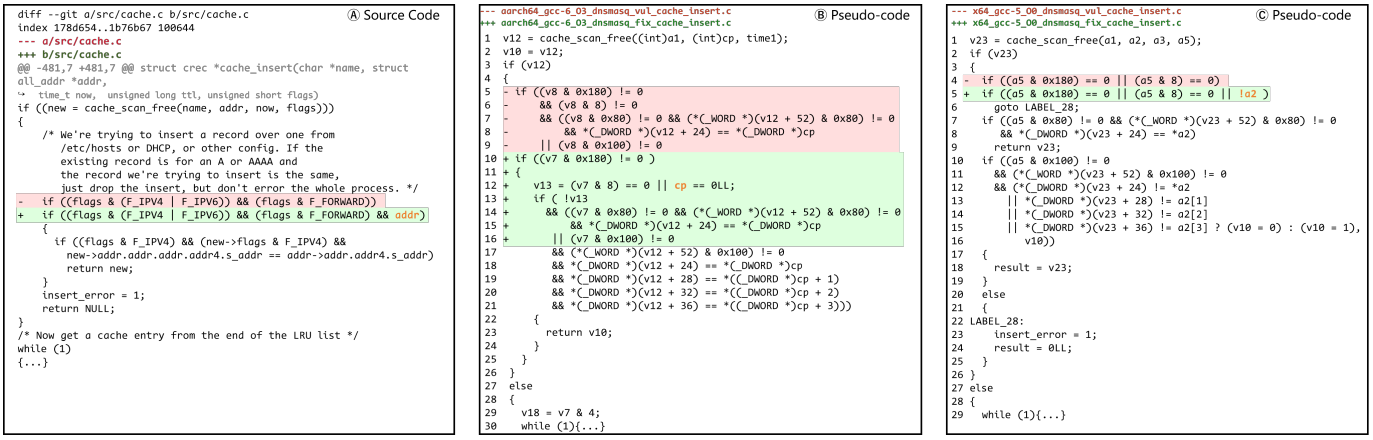

Fig. 1. The figure illustrates the patch for CVE-2015-8899, a tiny-patch vulnerability in Dnsmasq 2.75. It compares decompiled pseudo-code from vulnerable and patched binaries under different compilation settings, highlighting structural differences while preserving core semantics.

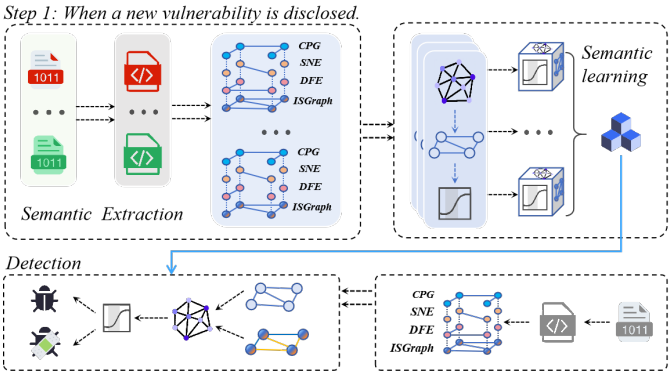

Fig. 2. Overall framework of ISGraphVD.

for graph-based vulnerability detection can also be applied to other languages.

B. Identifier Sensitive Graph

We perform a variable-level analysis on pseudo-code and trace variable value paths to uncover finer-grained semantic information. We define four types of dependencies—**read-Tracking** (E_{rt}), **read-after-write** (E_{raw}), **write-after-read** (E_{war}), and **write-after-write** (E_{waw})—based on the sequence of variable read and write operations. We introduce intra-block and inter-block dependencies to capture the control semantics introduced by the control flow structure and to address long-range dependencies. Finally, we incorporate these dependencies into the ISGraph to effectively capture variable interactions and enhance the representation of data flow.

In CPG, nodes with the 'Identifier' attribute correspond to program variables. We represent the dependencies between variables as edges that connect the nodes corresponding to these variables' identifiers, thereby constructing the ISGraph. The construction process is as follows: Variables in the code are ordered top-to-bottom and left-to-right, defined as $X = \{x_1, x_2, \ldots, x_n\}$, where for each $x_i \in X$, the k-th variable with the same name is x_i^k. Given the pseudo-code set $P = \{p_1, p_2, ..., p_k\}$, we first parse each p_i into a CPG. The CPG comprises node set V and edge set E_{cpg}. We model dependencies as edges and categorize them into different-name edges (E_{dne}) and same-name edges (E_{sne}), depending on whether variables share the same name. These edges combine with the CPG to form the ISGraph $G = (V, E)$, where $E = E_{cpg} \cup E_{dne} \cup E_{sne}$.

Variable-level dependencies First, we introduce the variable-level dependency relationship set $E_{dep} = E_{raw} \cup E_{war} \cup E_{rt}$ for same-name variables and different-name variables both within and across statements. If a variable x_i^k is in the read state, its value comes from x_i^{k-1}. Depending on whether x_i^{k-1} is in the read or write state, we establish the two variable-level dependency relationships E_{rt} and E_{raw} between x_i^k and x_i^{k-1}:

$$E_{rt} = \{\langle x_i^k, x_i^{k-1}\rangle \mid x_i^k \text{ is in read state}, \; x_i^{k-1} \text{ is in read state}\}$$

$$E_{raw} = \{\langle x_i^k, x_i^{k-1}\rangle \mid x_i^k \text{ is in read state}, \; x_i^{k-1} \text{ is in write state}\}$$

If a variable x_i^k in statement s is in the write state, its value is assigned by other variables in the same statement, denoted as the set X_s. The dependency relationship E_{war} is defined as follows:

$$E_{war} = \{\langle x_i^k, x_j^d\rangle \mid x_i^k, x_j^d \in X_s, x_i^k \text{ in write state}, \; x_j^d \text{ in read state}\}$$

We reorder variables in the semantic context based on E_{dep}. For two variables connected by a dependency $e \in E_{dep}$, the variable read or written later is placed after the one read or written earlier.

Intra-block and Inter-block dependencies Inter-block dependencies arise from relationships among variables in critical zones, such as block entry and exit points. We define two sets: $B_{in} = \{x_i \mid x_i \in X, x_i \text{ is the first variable in a block}\}$ and $B_{out} = \{x_i \mid x_i \in X, x_i \text{ is the last variable in a block}\}$. The

terms "first" and "last" are based on the semantic context order. We will discuss intra-block and inter-block dependencies in detail, categorizing them by control flow structures to analyze dependencies within and across blocks.

Sequential structures imply that code is executed line by line and are the most basic and common structures. Intra-block dependencies typically occur within these structures. We consider two cases based on whether variables are in the same statement: ① For variables in the same statement, dependencies arise from operations such as assignment, arithmetic, logical, relational operations, function calls, and array/list accesses. These behaviors introduce two types of variable dependencies: E_{war} and E_{rt}. ② Between different statements, we only consider data dependencies among variables with the same name, including E_{rt} and E_{raw}.

The *selection structure* executes different code blocks based on a condition's truth value, with common examples being *if* and *switch*. This structure includes multiple branch blocks, each with a conditional judgment and an execution block. Inter-block variable dependencies in these structures can be categorized as follows: ① A variable $x_i^k \in B_{\text{out}}$ from the basic block before the selection structure may have E_{rt} and E_{raw} dependencies with $x_i^{k+n} \in B_{\text{in}}$ from branch blocks. Dependencies E_{rt} and E_{raw} arise if x_i^{k+n}, which is in the read state, is in the conditional or execution block, and x_i^k is in read or write state. ② A variable $x_i^k \in B_{\text{out}}$ in a branch's execution block may have E_{rt} dependencies with $x_i^{k+n} \in B_{\text{in}}$ in the branch's conditional judgment block. ③ If $x_i^{k+n} \in B_{\text{in}}$ in the basic block after the selection structure is in a read state, it may have E_{rt} and E_{raw} dependencies with $x_i^k \in B_{\text{out}}$ in multiple branch blocks.

The *loop structure* executes code blocks repeatedly, including structures like while, for, and do-while. It consists of a conditional judgment block and a loop basic block. There are three types of inter-block dependencies within this structure: ① The variable $x_i^k \in B_{\text{out}}$ from the block preceding the loop may have E_{rt} and E_{raw} dependencies with $x_i^{k+n} \in B_{\text{in}}$ in the conditional judgment block. ② During loop iterations, $x_i^k \in B_{\text{out}}$ in the loop block may have E_{rt} or E_{raw} dependencies with $x_i^{k+n} \in B_{\text{in}}$ in the conditional judgment block. ③ The variable $x_i^k \in B_{\text{out}}$ in the loop block may have E_{rt} and E_{war} dependencies with $x_i^{k+n} \in B_{\text{in}}$ in the block after the loop, depending on whether x_i^k is in the read state.

The *Jump structures* alter code execution order, including statements like break and continue. These statements divide their blocks into segments before and after the control flow statement, creating new inter-block dependencies. For instance, a continue statement, which skips to the next iteration, introduces an E_{raw} or E_{rt} dependency between $x_i^k \in B_{\text{in}}$ and $x_i^{k+n} \in B_{\text{out}}$ in the block preceding the continue statement.

C. ISGraphVD Checker

1) Detection Model: By introducing a cross-graph attention matching mechanism, GMN [17] can be more sensitive to the difference between graphs. Given a pair of graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the objective of GMN is to calculate the similarity $s(G_1, G_2)$ between them. The model includes three parts: Encoder, Propagator, and Aggregator. Each graph $G = (V, E)$ is characterized by sets of nodes V and edges E. A feature vector x_i represents each node $i \in V$, and a feature vector x_{ij} represents each edge $(i, j) \in E$.

The node and edge vectors are initialized independently through distinct multilayer perceptrons (MLPs) in the encoding layer. GMN's propagation layer not only aggregates information $\sum_j m_{j\to i}$ between nodes in the same graph but also utilizes cross-graph matching information $\mu_{k\to i}$ between nodes in the two graphs involved in similarity calculation to update node features, as shown in Equation 1.

$$h_i^{(t+1)} = f_{\text{node}}(h_i^{(t)}, \sum_j m_{j\to i}, \sum_{k'} \mu_{k'\to i}) \tag{1}$$

$$m_{j\to i} = f_{\text{message}}(h_i^{(t)}, h_j^{(t)}, e_{ij}), \quad \forall (i,j) \in E_1 \cup E_2 \tag{2}$$

$$\begin{aligned} \mu_{k\to i} &= f_{\text{match}}(h_i^{(t)}, h_k^{(t)}), \\ &\forall i \in V_1, k \in V_2 \text{ or } i \in V_2, k \in V_1 \end{aligned} \tag{3}$$

Specifically, for $\forall i \in V_1$, initially, the attention weights $a_{k\to i}$ between $\forall k \in V_2$ and i are computed based on their similarity, as depicted in Equation 4. This attention weights $a_{k\to i}$ becomes larger as the similarity between the two nodes increases. Then, the difference between the nodes belonging to the two graphs is calculated using Equation 6. Finally, the cross-graph matching information $\sum_k \mu_{k\to i}$, which is used to update node i, is derived by aggregating the matching information between each node in V_2 and node i in V_1.

$$a_{k\to i} = \frac{\exp(s_h(h_i^{(t)}, h_k^{(t)}))}{\sum_{k'} \exp(s_h(h_i^{(t)}, h_{k'}^{(t)}))} \tag{4}$$

$$\mu_{k\to i} = a_{k\to i}(h_i^{(t)} - h_k^{(t)}) \tag{5}$$

$$\begin{aligned} \sum_k \mu_{k\to i} &= \sum_k a_{k\to i}(h_i^{(t)} - h_k^{(t)}) \\ &= h_i^{(t)} - \sum_k a_{k\to i} h_k^{(t)} \end{aligned} \tag{6}$$

After T rounds of propagation, we obtain the node representations for the nodes in both graphs. We then use a function f_G to derive the graph representations from these node representations, as described in 7. The distance between the graph representations can then be computed using a distance function.

$$f_G = MLP_G(\sum_{v_i \in V} \sigma(MLP_{gate}(h(T)_i) \odot MLP(h(T)_i))) \tag{7}$$

2) Implementation: The procedures involved in training and detection are different, and we will introduce them separately.

Training When training, the model requires paired inputs. For each CVE, we partition ISGraphs into two sets based on whether they are vulnerable or patched, denoted as $ISGraph_{vul}$ and $ISGraph_{ptd}$. For any G_i and G_j belonging

to $ISGraph_{vul}$, where G_i is not equal to G_j, we pair them to form similar (positive) pairs with a label of 1. For any G_i in $ISGraph_{vul}$ and any G'_i in $ISGraph_{ptd}$, we pair them to form dissimilar (negative) pairs with a label of 0. We use a margin-based loss function defined as:

$$L_{\text{pair}} = E_{(G_1,G_2,t)}\left[\max\left\{0, \gamma - t\left(1 - d(G_1, G_2)\right)\right\}\right] \quad (8)$$

where $t \in -1, 1$ is the pair's label, $d(G_1, G_2)$ is the Euclidean distance between the graphs, and $\gamma > 0$ is the margin parameter. This function encourages positive pairs to have smaller distances $(d(G_1, G_2) < 1 - \gamma)$ and negative pairs to have larger distances $(d(G_1, G_2) > 1 + \gamma)$.

Detection For each CVE, we choose a baseline vulnerable graph G_{base} and a threshold t. When a graph G_d is under detection if the distance between G_d and G_{base} is smaller than t, we infer that G_d contains the vulnerability; otherwise, we conclude that it does not contain the vulnerability.

We choose G_{base} from the training set and then determine the threshold t accordingly. We use "shortest average distance" to select G_{base}. We denote the vulnerable and patched graphs in the training set as S_{vul} and S_{ptd}. For each vulnerable graph $G_{\text{vul}} \in \mathcal{S}_{\text{vul}}$, the average distance between G_{vul} and the other graphs in the set $\mathcal{S}_{\text{vul}}$ is calculated as follows:

$$D_{\text{avg}}(G_{\text{vul}}) = \frac{1}{|\mathcal{S}_{\text{vul}}|} \sum_{G_i \in \mathcal{S}_{\text{vul}}} \text{dist}(G_{\text{vul}}, G_i) \quad (9)$$

And we can choose the G_{base} following the formula:

$$G_{\text{base}} = G_i \mid \min(D_{\text{avg}}(G_i)), \quad G_i \in \mathcal{S}_{\text{vul}} \quad (10)$$

For each $G_{vul} \in S_{vul}$ and $G_{ptd} \in S_{ptd}$, we calculate the distances of each (G_{vul}, G_{base}) pair and (G_{ptd}, G_{base}) pair and determine t by the distribution of distances. Since most samples are negative in the real world, having a low false-positive rate is of great significance to the actual performance of our method. Therefore, we choose to set the lowest possible threshold.

IV. Experimental Evaluation

We evaluate the effectiveness of ISGraphVD following six research questions.

- RQ1: Can ISGraphVD maintain high accuracy across different compilation settings?
- RQ2: Can ISGraph effectively improve the accuracy of detecting tiny-patch vulnerabilities?
- RQ3: Does ISGraphVD make false alarms frequently?
- RQ4: Does the one-model-per-CVE strategy improve vulnerability detection accuracy?
- RQ5: Is the time cost of ISGraphVD acceptable?
- RQ6: What about the performance of ISGraphVD in the real-world dataset?

A. Implementation

We have developed a prototype of ISGraphVD using Python 3.8.5. For binary file decompilation, we utilize IDA Pro 8.3. Since the Joern [25] can only produce CPG for pseudo-code, we extend it using Scala scripts to extract dependency edges between variables from pseudo-codes and merge these edges into CPG to form ISGraph. The model is implemented in PyTorch (v1.11.0).

B. Experiment Setup

1) Runtime Environments: All the experiments are conducted on a Rocky Linux 9.2 server with an Intel(R) Xeon(R) Gold 6348 CPU running at 2.60GHz, 512 GB RAM, and 4 NVIDIA A100-PCIE-40GB GPU cards. The deep learning architecture is built on the NVIDIA CUDA Toolkit 12.2.

2) Datasets: Existing binary datasets [26], [27] are typically compiled with the same options, so they can not reflect the ability to generalize across different architectures, compilers, and optimization levels. We use the dataset proposed by Liu et al. [15] to construct Dataset I to evaluate our method in this paper, as shown in Table I. It contains only eight vulnerabilities from five open-source projects (dnsmasq, curl, miniupnp, busybox, and hostapd) and include four common vulnerability types. Moreover, the vulnerabilities are categorized into four levels based on the modification ratio: tiny (the modified code accounts for less than 2% of the total function code, the same below), small $(2\% \sim 10\%)$, medium $(10\% \sim 20\%)$ and large $(>= 20\%)$. These eight vulnerabilities encompass all modification levels. To reflect the imbalanced distribution of real-world (Dataset I is a balanced dataset), with the minority of samples being labeled as vulnerable, We construct **Dataset II** using **Dataset I** for evaluating the FPR of our method. Specifically, for each vulnerability V in Dataset I, the vulnerable samples associated with V are labeled as vulnerable, while all other samples in Dataset I are considered non-vulnerable.

To further validate the effectiveness of our approach in both typical scenarios and when dealing with tiny-patch vulnerabilities, we randomly selected eight vulnerabilities, including two tiny-patch vulnerabilities, to construct **Dataset III**. To improve diversity, we additionally introduced OpenSSL as a new component alongside the previous open-source projects. To further evaluate the generalizability of our approach, we intentionally used different compiler versions compared to Dataset I when building this dataset. Fig.3 illustrates the process of **building the dataset for each CVE detection model**. First, we collect CVE details and patch information (pre- and post-patch commits) from the CVE [28] and National Vulnerability Database (NVD) [29], then download the corresponding vulnerable and patched versions of the source code from the relevant Git repositories. The most challenging part is compiling all the related source code files from different open-source software repositories. To address this, we manually set up the build environments and determine the correct compilation commands. Next, we cross-compile these two versions of the source code using various architectures (X86, X64, MIPS, MIPSel, ARM, ARMhf, ARM64), compilers (GCC-11, GCC-12, GCC-13, CLANG-10, CLANG-11, CLANG-12), and optimization levels (O0-O3, Os, default), resulting in $2 \times 7 \times 6 \times 6 = 504$ different binaries. Notably, although different optimization levels typically produce different binary files, there are specific

TABLE I
VULNERABILITIES OF DATASET I, II AND III AND TRAINING RESULTS OF ISGRAPHVD

Dataset	CVE ID	Type	Modification Level	Project	AUC(Training)	AUC(Validation)
Dataset I & Dataset II	2021-22901	RCE	Medium (11.11%)	cURL	100.00%	100.00%
	2019-16275	DoS	Small (2.35%)	HostAP	100.00%	100.00%
	2019-5482	DoS	Tiny (1.18%)	cURL	100.00%	100.00%
	2018-20679	Info Leak	Medium (12.82%)	Busybox	100.00%	100.00%
	2017-14491	RCE	Large (22.22%)	Dnsmasq	100.00%	100.00%
	2017-1000494	Overflow	Large (25.00%)	MiniUPnP	100.00%	100.00%
	2015-8899	DoS	Tiny (0.64%)	Dnsmasq	100.00%	100.00%
	2015-6031	RCE	Large (20.00%)	MiniUPnP	100.00%	100.00%
Dataset III	2023-46219	Unencrypted Data	Large (27%)	cURL	100.00%	100.00%
	2021-3448	Standard Security Check	Medium (5.9%)	Dnsmasq	100.00%	100.00%
	2019-12110	DoS	Tiny (1.96%)	MiniUPnP	99.99%	99.99%
	2017-14496	Integer Underflow	Small (9.92%)	Dnsmasq	100.00%	100.00%
	2017-13704	Improper Input Validation	Tiny (0.3%)	Dnsmasq	100.00%	100.00%
	2023-0401	NULL Pointer Dereference	Large (20.4%)	Openssl	99.99%	99.99%
	2023-0216	NULL Pointer Dereference	Large (37.5%)	Openssl	100.00%	100.00%
	2023-0217	NULL Pointer Dereference	Large (21.4%)	Openssl	100.00%	100.00%
Average					99.99%	99.99%

cases—such as simple code structures, particular compiler behaviors, or ineffective optimizations—where the binaries compiled under different optimization levels may be identical. To ensure a practical and representative evaluation of the model's generalization on unseen binaries, we **deduplicate** these binaries, ensuring that no training samples appear in the test set.

Dataset IV consists of 121 real-world firmware samples from eight vendors (ASUS, D-Link, NetGear, TP-Link, NetCore, WD, EDIMAX, and Hanfeng), from which we extracted 208 binaries. We utilize this dataset to evaluate the real-world performance of ISGraphVD and assess the impact of supply chain vulnerabilities.

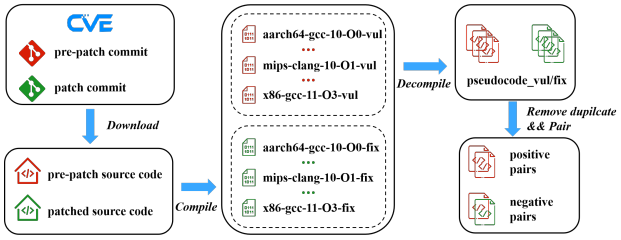

Fig. 3. The process of building the dataset.

3) Evaluation Metrics: Since determining whether a function contains a known CVE-related vulnerability is a binary classification task, we select the following performance metrics:

- **Accuracy** measures the ratio between the number of samples correctly classified by the classifier and the total number of samples: $ACC = \frac{TP+TN}{TP+FP+FN+TN}$.
- **Precision** refers to the case where the method accurately identifies a vulnerable binary: $P = \frac{TP}{TP+FP}$.
- **Recall** is the proportion of actual vulnerable binaries detected to all vulnerable binaries: $R = \frac{TP}{TP+FN}$.
- **F1** Score represents the overall performance of the test, calculated as the harmonic mean of precision and recall: $F1 = 2 \times \frac{P \times R}{P+R}$.

We use Area Under Curve (AUC) to evaluate ISGraphVD, representing the area under the Receiver Operating Characteristic (ROC) curve, where a higher AUC value indicates better classifier performance.

C. Performance Evaluation

Since real-world binary compilation settings are often unknown, we adopt XM, proposed in [30], as the evaluation task and the benchmark for comparative experiments. XM simulates diverse compilation settings by varying architectures, bitness, compilers, compiler versions, and optimization levels, making it well-suited to reflect practical detection scenarios.

1) Training and Detection Performance: To evaluate ISGraphVD's performance, we train the models using Dataset I and III, with a train/validation/test split of 60%/20%/20%. During training, we set the following parameters: Adam optimizer, a node feature dimension of 56, a graph representation dimension of 128, and a learning rate of either 0.0001 or 0.00005. The lower learning rate (0.00005) is typically used for vulnerabilities with tiny modifications to ensure stable convergence. We set the maximum number of training epochs to 100, with a minimum of 10. If the model converges early, training stops before reaching 50 epochs. The results are presented in Table I, where we observe that all the models trained for the selected CVEs successfully converge.

Before detection, we must select the baseline graph G_{base} and determine the threshold t as described in Section III-C2. Fig. 4 shows that clear boundaries can be established to determine the threshold for each type of modification level. The red and green points represent the distances between the positive and negative samples in the training set and G_{base}. Although some models converge within a single epoch, their loss has not yet approached zero. Continuing the training process enables the models to better distinguish between vulnerable and patched samples. As shown in Table II, the average accuracy of ISGraphVD is 99.47%, the average recall is 98.75%, the average precision is 100.00%, and the average F1-Score is 99.39%. Thus, we can answer RQ1, confirming

that ISGraphVD achieves high-accuracy vulnerability detection even across different compilation settings.

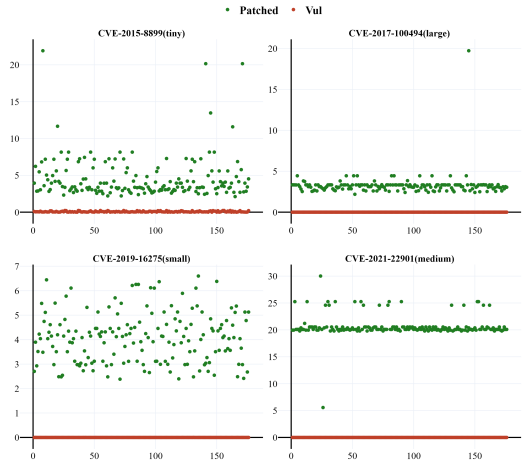

Fig. 4. The process of determining the thresholds for each modification level.

TABLE II
EVALUATION RESULTS USING TEST SET IN DATASET I AND III

CVE ID	Accuracy	Recall	Precision	F1-Score
2021-22901	100.00%	100.00%	100.00%	100.00%
2019-16275	100.00%	100.00%	100.00%	100.00%
2019-5482	100.00%	100.00%	100.00%	100.00%
2018-20679	100.00%	100.00%	100.00%	100.00%
2017-14491	100.00%	100.00%	100.00%	100.00%
2017-1000494	100.00%	100.00%	100.00%	100.00%
2015-8899	98.48%	97.01%	100.00%	98.48%
2015-6031	100.00%	100.00%	100.00%	100.00%
2023-46219	99.00%	98.00%	100.00%	99.00%
2021-3448	97.14%	94.12%	100.00%	96.97%
2019-12110	99.00%	98.00%	100.00%	99.00%
2017-14496	99.10%	98.25%	100.00%	99.12%
2017-13704	98.00%	96.00%	100.00%	98.00%
2023-0401	97.56%	95.12%	100.00%	97.52%
2023-0216	100.00%	100.00%	100.00%	100.00%
2023-0217	98.96%	97.92%	100.00%	98.95%
Average	**99.47%**	**98.75%**	**100.00%**	**99.39%**

To answer RQ2, we use CPG as the baseline to evaluate whether ISGraph improves vulnerability detection accuracy, particularly in tiny-patch vulnerabilities. As shown in Table III, ISGraph consistently achieves higher detection performance than CPG across all evaluated CVEs. The improvement is especially pronounced for tiny-patch vulnerabilities—CVE-2019-5482, CVE-2015-8899, CVE-2019-12110, and CVE-2017-13704—where ISGraph yields an average accuracy gain of approximately 4%. This can be attributed to the superior graph feature representation of ISGraph, which fully captures finer-grained semantic features. As a result, ISGraphVD proves effective in detecting tiny-patch vulnerabilities and maintains strong performance across various modification levels.

In addition, we conduct a visualization study of the model's attention scores. In GMN, the cross-graph attention scores $a_{k\to i}$ in Equation 6 measure the similarity between nodes from different code fragments. After training, we assume that node pairs with similar semantics and context have higher attention values than others. In Fig. 5, we highlight the five highest attention values $a_{k\to i}$ from the entire attention matrix and illustrate their corresponding positions in the pseudo-codes. We choose CVE-2015-6031 for demonstration. The two pseudo-codes on the figure's left and right sides are obtained by decompiling the binary files compiled from the vulnerability source code with different cross-compilation options (mipsel-clang-7-default and x86-gcc-6-Os). The lines represent connections between node pairs exhibiting the highest similarities between the left and right pseudo-codes. While certain attention links may be intuitive to humans, some connections remain difficult to interpret, such as the edge extending from the literal node "2436" in the left code to the literal node "0" in the right code. Furthermore, we observe that as the modification ratio decreases, the model converges more slowly, indicating that the modification level is the primary factor affecting training performance.

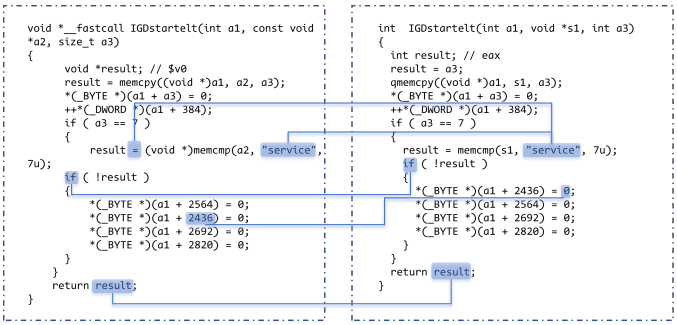

Fig. 5. The corresponding locations in pseudo-codes for the node pairs with the five highest attention scores in a (vul,vul) pair.

2) False Positive Evaluation: In the real world, the majority of samples are negative, meaning they do not contain vulnerabilities. Incorrectly classifying non-vulnerable samples as vulnerable requires additional manual effort to verify these false positives, significantly increasing the workload. Therefore, for a method to be practically usable, it is crucial to maintain a low FPR. However, many existing studies in this research field focus solely on whether the model can accurately identify the patch status, overlooking false alarms in irrelevant functions.

This section evaluates whether ISGraphVD can achieve a low FPR while maintaining high accuracy. We utilize the vulnerability-irrelevant samples in Dataset II to assess the FPR of ISGraphVD. As shown in Table IV, ISGraphVD achieves an average FPR of 0.00%, demonstrating its capability to distinguish vulnerable functions from unrelated ones, even when trained solely on vulnerable and patched functions. Through this evaluation, we provide a satisfactory answer to RQ3, confirming that ISGraphVD meets real-world demands and is suitable for large-scale vulnerability detection.

3) Evaluation Against Shared-Model Approaches: To answer RQ4, we compare ISGraphVD with the open-source implementations of existing studies, including Gemini,

TABLE III
COMPARATIVE RESULTS OF CPG AND ISGRAPH ON VULNERABILITY DETECTION

CVE ID	**2019-5482**	2017-14491	2017-1000494	2018-20679	**2015-8899**	2019-16275	2021-22901	2015-6031
CPG	97.43%	98.92%	100.00%	100.00%	91.34%	100.00%	100.00%	100.00%
ISGraph	100.00%	100.00%	100.00%	100.00%	98.48%	100.00%	100.00%	100.00%
CVE ID	2023-46219	2021-3448	**2019-12110**	2017-14496	**2017-13704**	2023-0401	2023-0216	2023-0217
CPG	93.41%	93.14%	91.38%	95.40%	91.25%	91.46%	94.79%	92.71%
ISGraph	99.00%	97.14%	99.00%	99.10%	98.00%	97.56%	100.00%	98.96%

TABLE IV
FPR EVALUATION USING DATASET II

CVE ID	Total Samples	FP	FPR
2021-22901	2,711	0	0.00%
2019-16275	2,514	0	0.00%
2019-5482	2,497	0	0.00%
2018-20679	2,518	0	0.00%
2017-14491	2,543	0	0.00%
2017-1000494	2,662	0	0.00%
2015-8899	2,427	0	0.00%
2015-6031	2,648	0	0.00%
Average	2,565	**0**	**0.00%**

TABLE V
COMPARATIVE EVALUATION RESULTS USING DATASET I

CVE ID	Gemini	VulSeeker	FIT	ISGraphVD
2019-5482	49.56%	51.55%	93.55%	100.00%
2017-14491	71.10%	45.50%	93.98%	100.00%
2017-1000494	72.16%	43.19%	94.14%	100.00%
2018-20679	50.11%	51.58%	93.83%	100.00%
2015-8899	50.53%	51.04%	89.12%	98.48%
2019-16275	50.12%	49.44%	95.13%	100.00%
2021-22901	53.98%	54.93%	94.27%	100.00%
2015-6031	48.09%	45.91%	90.00%	100.00%
Average	**56.79%**	**49.60%**	**93.43%**	**99.73%**

VulSeeker, and FIT. All these methods are graph-based approaches designed for detecting binary code similarity at the function level, and they typically use a single shared model to learn features from multiple CVEs. We use Dataset I to assess these approaches, applying the recommended best parameters from their respective papers and ensuring sufficient training epochs. Since the papers of these approaches lack sufficient details on threshold determination, we can only use AUC for comparison with ISGraphVD. Additionally, due to constraints in the open-source implementations of Gemini and FIT, our evaluation is limited to three architectures: ARM64, X64, and MIPS.

The results in Table V demonstrate that ISGraphVD outperforms the other three methods across all vulnerabilities. For the tiny-patch vulnerabilities CVE-2015-8899 and CVE-2019-5482, Gemini and VulSeeker achieve an AUC of around 50%, while FIT achieves significantly higher results, at 89.12% and 93.55%, respectively. However, these AUC values remain at least 6% lower than those of ISGraphVD. This performance gap persists across all modification levels, where ISGraphVD consistently achieves the highest accuracy among the compared methods. This improvement can be attributed to the one-model-per-CVE strategy, which isolates the learning process for each vulnerability, thereby preventing feature interference and enabling more precise representation of vulnerability-specific patterns.

D. Runtime Efficiency

We evaluate the efficiency of ISGraphVD by measuring the time required for preprocessing, model training, and detection. Preprocessing mainly involves four steps: cross-compilation, decompilation, and graph generation. Due to the variable and challenging nature of source code collection, cross-compilation, and decompilation, we focus our evaluation on graph generation time (using 1 thread), model training time (using 1 thread and 1 GPU), and vulnerability detection time (using 1 thread and 1 GPU). Table VI presents the time costs of our approach. Due to the multi-model architecture of our method, the training cost is higher than that of existing works. From the start of training until full convergence, the training process typically takes around 100 minutes for most CVEs. However, a notable exception is CVE-2015-8899, which requires 100 epochs to achieve optimal performance. By monitoring the loss at each epoch, we observed that the model's validation set AUC approached 99% by the 20th epoch. Despite this, the test set accuracy remained around 95%, still exceeding PG-VulNet's best accuracy of 91.34%. We found that continuing training beyond this point further improved detection accuracy and enhanced model stability. Therefore, we prioritized accuracy over training time, considering the cost acceptable since it is a one-time process.

Detection time Our method takes approximately 1.39 seconds to complete one detection, with around 1.23 seconds for graph preparation and 0.16 seconds for detection, which is slightly better than existing methods [16]. Additionally, a 1.39-second processing time for determining the presence of a vulnerability in a specific binary is acceptable for practical use. Thus, this serves as our answer to RQ5.

E. Vulnerability Detection in Real-World IoT Firmware

To answer RQ6, we evaluate ISGraphVD on the known vulnerability detection task for IoT device firmware, using the real-world Dataset IV. Among 208 binaries, ISGraphVD finds 62 vulnerabilities, as shown in Table VII. We manually

TABLE VI
TIME COST OF OUR APPROACH USING DATASET I AND III

CVE ID	Graph(s)	Train(s)	Detect(s)
2021-22901	1.26	147.08	0.19
2019-16275	1.08	8899.11	0.30
2019-5482	1.14	9916.04	0.12
2018-20679	1.00	3272.04	0.07
2017-14491	1.14	2412.06	0.19
2017-1000494	1.09	1397.79	0.19
2015-8899	1.15	37824.33	0.02
2015-6031	1.14	653.78	0.08
2023-46219	1.11	723.26	0.17
2021-3448	1.35	9359.62	0.24
2019-12110	1.09	1584.57	0.19
2017-14496	1.19	12779.85	0.14
2017-13704	1.78	14458.98	0.24
2023-0401	1.12	1473.34	0.13
2023-2016	1.13	1398.56	0.13
2023-2017	1.10	1331.23	0.12
Average	**1.23**	**7,771.82**	**0.16**

reviewed the binaries where vulnerabilities were detected and made the following observations. (1) Although we only use pre-patch commit and patch commits for training, ISGraphVD can still accurately determine whether a version before the pre-patch commit or after the post-patch commit contains vulnerabilities. However, for three vulnerabilities (CVE-2021-22901, CVE-2019-16275, and CVE-2019-5482), the model did not detect any vulnerable binaries. We found that this was because there was no overlap between the affected versions of these vulnerabilities and the versions of the open-source components used in the firmware. For instance, the affected version range for CVE-2021-22901 is 7.75.0 to 7.76.1. However, all versions of libcurl.so used in the firmware dataset were earlier than 7.75.0, explaining why no vulnerable binaries were found. For CVE-2015-8899, ISGraphVD correctly identified most of the vulnerable binaries. However, three false positives were observed, all belonging to the same Asus firmware series, which reused the same version of dnsmasq, leading to identical pseudo-code. Manual verification revealed that these samples were similar to the false positives observed during model testing, highlighting an area for future improvement. (2) Our findings indicate that third-party library vulnerabilities remain a major security concern in IoT firmware, as known vulnerabilities in firmware images are often not patched in time. CVE-2017-1000494 and CVE-2017-14491 affected nearly all vendors in the dataset, with Western Digital's personal cloud products being the most impacted, covering four different series. This evaluation underscores the critical importance of detecting known vulnerabilities to enhance the security of IoT devices.

V. LIMITATION

Although our method effectively addresses the challenges posed by cross-architecture and tiny-patch vulnerabilities, it still cannot handle the following scenarios:

TABLE VII
REAL WORLD DETECTION RESULTS

CVE ID	Binary	Total	Vulnerable	FP
2021-22901	libcurl.so	41	0	0
2019-16275	hostapd	18	0	0
2019-5482	libcurl.so	41	0	0
2018-20679	busybox	56	9	0
2017-14491	dnsmasq	45	17	0
2017-1000494	miniupnpd	36	29	0
2015-8899	dnsmasq	45	0	3
2015-6031	miniupnpc	12	7	0

String change ISGraphVD cannot handle situations where vulnerabilities are fixed solely by modifying strings. For example, CVE-2022-27780, a vulnerability in cURL, is illustrated in Fig. 6, where the parameter of the function call 'strcspn' changed from "\r\n" to "\r\n\t/:#?!@". ISGraph extracts dependencies between variables where strings belong to the 'LITERAL' type. When converting ISGraph into the feature matrix, we only consider the nodes' type, not their values. Therefore, changes in strings do not affect the feature matrix of nodes.

```
@@ -678,8 +678,8 @@ static CURLUcode hostname_check(struct Curl_URL *u, char *hostname)
#endif
  }
  else {
-   /* letters from the second string is not ok */
-   len = strcspn(hostname, " \r\n");
+   /* letters from the second string are not ok */
+   len = strcspn(hostname, " \r\n\t/:#?!@");
    if(hlen != len)
      /* hostname with bad content */
      return CURLUE_BAD_HOSTNAME;
```

Fig. 6. Patch of CVE-2022-27780.

Inlined function ISgraphVD determines the similarity between two functions by computing their semantic similarity. On the one hand, function inlining can alter semantics. On the other hand, the inline expansion may eliminate the vulnerable functions. Therefore, our approach currently cannot handle cases involving inlined functions.

In the future, we will test ISGraphVD on more diverse and challenging vulnerability samples to further explore its limitations and guide design improvements.

VI. RELATED WORK

A. Code Representation Learning

To improve code representation, researchers have evolved from using conventional token sequences [31] to abstract syntax trees (ASTs) [32]–[35] and ultimately to graphs [36]–[38]. ASTs provide rich syntactic information for modeling source code, as seen in approaches like tree-based CNNs for code classification [34] and Tree-LSTM for code clone detection [39] . However, while ASTs capture syntax well, they lack semantic details like control and data flow. To address this, some studies introduce additional edges on ASTs, use control flow graphs (CFGs), or data flow graphs (DFGs) to enhance semantic representation. For instance, FA-AST [40] includes explicit control and data flow edges, and GraphCodeBERT [41] leverages data flow for efficient semantic representation. DeepSim [42] and Gemini [18] use CFGs to

detect code clones, capturing semantic features and binary code similarity, respectively.

B. Learning-based Binary Code Similarity Detection

In learning-based binary code similarity detection technology, the utilization of high-dimensional feature vectors generated through machine learning imparts greater robustness to code variations arising from cross-architectural differences, leading to increased accuracy. There are many works based on function granularity, similar to ISGraphVD. Genius [10] applies clustering algorithms to generate high-dimensional representations of Attributed Control Flow Graphs (ACFGs) at the basic block level, leveraging statistical and structural features for efficiency. Gemini [18] builds on Genius by introducing an end-to-end graph neural network, using a Siamese Network [43] to compute function similarity scores. VulSeeker [7] further enhances this by incorporating semantic information, constructing Labeled Semantic Flow Graphs (LSFGs) from control and data flow graphs, and extracting semantic features via a deep neural network. However, both Gemini and VulSeeker rely on selecting the most similar functions from a candidate set, yet they cannot guarantee whether the highest-ranked function is truly similar to the target function. The common workaround is to return the top-K most similar functions as results. Some studies [5], [44] highlight the dependency on expert-defined control flow graph features, which may introduce biases. As a result, they propose directly extracting function features from raw binary code.

VII. Conclusion

In this paper, we proposed ISGraphVD, a novel approach for binary vulnerability detection that addresses challenges in cross-compilation environments and minimal patch modifications. By adopting a one-model-per-CVE strategy, ISGraphVD ensures independent learning of vulnerability features, enhancing generalization and scalability. Additionally, we introduced Identifier Sensitive Graph (ISGraph) to capture fine-grained variable dependencies and integrated a GMN with a cross-graph attention mechanism to focus on vulnerability-critical subgraphs, improving detection accuracy, particularly for Tiny-Patch vulnerabilities. Experimental results show that ISGraphVD outperforms state-of-the-art methods, demonstrating superior detection accuracy and robustness across compilation settings. Real-world experiments validate its practical applicability, and efficiency evaluations confirm that while the one-model-per-CVE strategy incurs higher initial training costs, it eliminates frequent retraining, making it a viable long-term solution. ISGraphVD provides a scalable and effective framework for binary vulnerability detection, offering significant advancements for securing IoT and other software ecosystems.

Acknowledgment

This work is supported by the National Key R&D Program of China under Grant No. 2024YFE0203800, HY-Project under No. 4E49EFF3, the National Natural Science Foundation of China under Grant No. 62302438, the Open Research Fund of the State Key Laboratory of Blockchain and Data Security, Zhejiang University, and the Supercomputing Center of Lanzhou University.

References

[1] P. Anand, Y. Singh, A. Selwal, M. Alazab, S. Tanwar, and N. Kumar, "Iot vulnerability assessment for sustainable computing: threats, current solutions, and open challenges," *IEEE Access*, vol. 8, pp. 168 825–168 853, 2020.

[2] B. Zhao, S. Ji, W.-H. Lee, C. Lin, H. Weng, J. Wu, P. Zhou, L. Fang, and R. Beyah, "A large-scale empirical study on the vulnerability of deployed iot devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1826–1840, 2020.

[3] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, "An investigation of the android kernel patch ecosystem," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3649–3666.

[4] H. Wang, Z. Gao, C. Zhang, M. Sun, Y. Zhou, H. Qiu, and X. Xiao, "Cebin: A cost-effective framework for large-scale binary code similarity detection," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 149–161.

[5] S. H. Ding, B. C. Fung, and P. 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)*. IEEE, 2019, pp. 472–489.

[6] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *arXiv preprint arXiv:2012.08680*, 2020.

[7] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 896–899.

[8] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 346–359.

[9] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi, and A. Hanna, "Binsign: fingerprinting binary functions to support automated analysis of code executables," in *ICT Systems Security and Privacy Protection: 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings 32*. Springer, 2017, pp. 341–355.

[10] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 480–491.

[11] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[12] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, 2017, pp. 79–94.

[13] N. Shalev and N. Partush, "Binary similarity detection using machine learning," in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, 2018, pp. 42–47.

[14] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the 46th ieee/acm international conference on software engineering*, 2024, pp. 1–13.

[15] X. Liu, Y. Wu, Q. Yu, S. Song, Y. Liu, Q. Zhou, and J. Zhuge, "Pg-vulnet: Detect supply chain vulnerabilities in iot devices using pseudo-code and graphs," in *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2022, pp. 205–215.

[16] H. Liang, Z. Xie, Y. Chen, H. Ning, and J. Wang, "Fit: Inspect vulnerabilities in cross-architecture firmware by deep learning and bipartite matching," *Computers & Security*, vol. 99, p. 102032, 2020.

[17] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*. PMLR, 2019, pp. 3835–3845.

[18] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. 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*, 2017, pp. 363–376.

[19] Hex-Rays, *IDA Pro*, Hex-Rays, Liège, Belgium, 2024, https://www.hex-rays.com/.

[20] National Security Agency (NSA), *Ghidra*, National Security Agency, Fort Meade, MD, USA, 2019, https://ghidra-sre.org/.

[21] L. Gao, Y. Qu, S. Yu, Y. Duan, and H. Yin, "Sigmadiff: Semantics-aware deep graph matching for pseudocode diffing," 2024.

[22] Y. Wang, P. Jia, X. Peng, C. Huang, and J. Liu, "Binvuldet: Detecting vulnerability in binary program via decompiled pseudo code and bilstm-attention," *Computers & Security*, vol. 125, p. 103023, 2023.

[23] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 590–604.

[24] WhiteSource Software, "The State of Open Source Vulnerabilities 2021," https://www.whitesourcesoftware.com/open-source-vulnerabilitymanagement-report/, 2021.

[25] J. D. Team, "Joern: Code property graph-based analysis," https://github.com/joernio/joern, 2024, accessed: 2025-08-18.

[26] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 887–902.

[27] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 376–387.

[28] The MITRE Corporation, "Common vulnerabilities and exposures (cve)," https://cve.mitre.org/, 2024, accessed: 2024-10-21.

[29] National Institute of Standards and Technology (NIST), "National vulnerability database (nvd)," Online, 2024, accessed: 2024-10-21. [Online]. Available: https://nvd.nist.gov/

[30] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2099–2116.

[31] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[32] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[33] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[34] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[35] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[36] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 2020, pp. 91–105.

[37] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[38] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and Polozov, "Generative code modeling with graphs," *arXiv preprint arXiv:1805.08490*, 2018.

[39] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[40] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[41] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[42] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 141–151.

[43] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," *Advances in neural information processing systems*, vol. 6, 1993.

[44] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 667–678.