Towards Generalizable Instruction Vulnerability Prediction via LLM-Enhanced Code Representation

Bao Wen, Jingjing Gu*, Jingxuan Zhang, Yang Liu, Pengfei Yu, and Yanchao Zhao
Nanjing University of Aeronautics and Astronautics, Nanjing, China. Email: {wenbao, gujingjing, jxzhang, liuyaaa, nuaaypf, yczhao}@nuaa.edu.cn

***Abstract*—Discovering potential vulnerabilities has long been a fundamental goal in software security. Among them, bit flips, caused by hardware or environmental disturbances, are increasingly recognized as a new type of vulnerabilities that threaten program reliability at the instruction level. However, existing work is often restricted to individual programs and requires retraining when applied to unseen code, severely limiting their practicality and responsiveness. In this paper, we propose CIVP, a novel framework for context-aware instruction vulnerability prediction, generalizing to unseen programs without retraining. Specifically, to capture the rich contextual semantics of instructions, CIVP first leverages Large Language Models (LLMs) to accurately extract semantic embeddings of instructions. Then, CIVP further constructs an instruction execution graph containing complex relations of program execution, which implicates the potential path of error propagation. To improve instruction representation for vulnerability prediction, CIVP enhances GraphSAGE with multi-hop diffusion to capture inter-program structural patterns and contextual dependencies, and adopts pseudo-labeling to improve the model's generalization for vulnerable instructions. Extensive experiments on a dataset of 26 real-world programs demonstrate that CIVP significantly outperforms the state-of-the-art approaches, achieving up to 20.5%↑ Recall and 18.5%↑ F1-score improvements. Notably, CIVP generalizes well to unseen programs, offering an efficient and scalable solution for proactive instruction-level hardening before software deployment.**

***Index Terms*—Instruction Vulnerability Prediction; Cross Program; Bit Flip**

I. Introduction

Software security is becoming increasingly crucial in real-world deployments, with the deep integration of software into critical infrastructure, transportation, and personal devices [1]. Recently, as system complexity and connectivity have increased, software security has faced a new problem—*Bit Flip, a bit of its memory instruction or register value is flipped during software execution.* For example, as shown in Fig.1, the instruction '*movl $1, %eax*' in the memory achieves the assignment of 1 to '*%eax*'. However, in extreme environments (e.g., lowering voltage) or security attacks, the immediate value '1' of the instruction may be flipped to '0' (i.e., *%eax=0*) or the last bit of '*%eax*' is flipped. Then, the error propagates with the execution of instructions, causing an error output (5 to 3) and affecting subsequent software execution. Besides, relevant studies have shown that 614,300 bit flips may occur globally every hour in computer systems, cell phones, and other devices [2]. Some technologies, such as Flip Feng Shui (FFS) [3], RowHammer attack [4] are expected to increase the likelihood of bit flips, which may cause unacceptable or catastrophic system failures (e.g., crash and silent data corruption[1]) by changing the software stack or instruction sequence of programs.

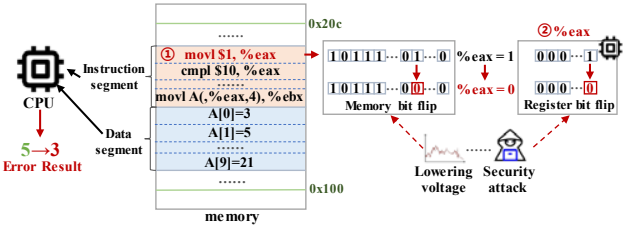


Fig. 1. **Schematic illustration of the generation and effects of bit flips during software execution. ① A bit flip that occurs in memory. ② A bit flip that occurs in registers.**

Although some techniques, including fault injection [5], [6], [7] and instruction vulnerability prediction [8], [9], have been proposed to detect the probabilities that instructions may cause program errors when affected by bit flips, they are still far from a solution. Specifically, fault injection aims to find vulnerable instructions by flipping each instruction bit, but suffers from the instruction explosion problem. Inspired by the great success of deep learning techniques, some learning-based approaches [10], [11] have been proposed to build an instruction vulnerability prediction model and achieve encouraging results. These approaches get some fault samples by performing partial fault injection on program instructions to train the model and identify vulnerable instructions. However, these approaches are limited to single programs and require retraining for new programs, which has a high overhead and lacks generality. This raises concerns about developing a generic model to achieve zero-shot instruction vulnerability prediction for new programs, an essential and undeveloped topic. Unfortunately, there are two main challenges:

Challenge 1: Lack of instruction semantics for enhancing program representation. Specifically, some approaches focus on manually designing heuristic features (e.g., operand, width) to predict instruction vulnerabilities. Unfortunately, these heuristic features do not represent the disparity of instruction semantics and are not always closely related to

* Corresponding author.

[1]https://research.facebook.com/research-awards/2022-silent-data-corruptions-at-scale-request-for-proposals/

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00177

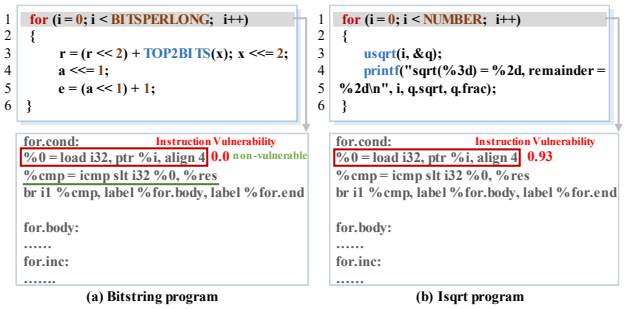


Fig. 2. **The *for* loop in different programs, while having the same form of instructions, exhibits different vulnerabilities.**

instruction vulnerabilities. The instructions with the same features may exhibit differences in vulnerabilities, especially across programs. Besides, they do not clarify the consistency of instruction semantics and the variability of data dependencies. For example, the instruction *'%cmp = icmp slt i32 %0, %res'* in Fig. 2(a), means comparing the result *'%0'* of *'load'* with *'%res'* and storing the result in *'%cmp'*, which is used by *'br'* instruction. Therefore, extracting robust instruction embeddings to represent programs fully is essential for instruction vulnerability prediction.

Challenge 2: Differences in the execution context of instructions in various programs. Due to the mixture of control and data dependencies, with the same instructions in different programs, the execution process may be deviant. Thus, their vulnerabilities also show differences. As shown in Fig. 2, the *'for'* loop in the "Isqrt" and "Bitstring" programs has the same form '*%0 = load i32, ptr %i, align 4, ...*' at the LLVM IR[2] level. However, the vulnerability of the *load* instruction in "Bitstring" is 0.0 (non-vulnerable), while "Isqrt" is 0.93. This is due to the differences in instruction sequence and data transfer during instruction execution, resulting in different operands for the *load* instruction. In contrast, most of the current approaches lack information on program structure and are limited to single programs. Thus, it is a challenging problem to model the whole path of instruction execution and fully mine the instruction context to solve the bias, which can provide better interpretability for error analysis.

To address the above challenges, we develop **CIVP**, a novel paradigm for ensuring software security against potential bit flips. It takes the program as input and calculates the probability that bit flips occurring in each instruction will cause an error in the result. Specifically, ① **Addressing Challenge 1**: To mine the contextual semantics of instructions, we first use the Large Language Model (LLM) to extract the semantic embeddings of instructions automatically. ② **Addressing Challenge 2**: Then, we notice that errors of bit flips propagate between instructions, along with control execution and data transfer. Thus, we extract the Instruction Execution Graph (IEG) from the IR code, which can be regarded as a combination of Control Flow Graph (CFG), Data Flow Graph (DFG), Call Graph (CG), and basic block. They can fully represent the process of instruction execution in a program and provide more refined information for locating vulnerable instructions. To learn better instruction representation for vulnerabilities, we use GraphSAGE [12] to summarize the knowledge of inter-program commonalities and enhance contextual heterogeneity between instructions with multi-hop diffusion. After that, we employ Pseudo-Labeling [13] to improve the model's generalization for vulnerable instructions to cope with the scarcity of fault samples and imbalanced distribution.

③ **Experiment**: Based on the fault injection tool LLTFI [14], we construct a dataset (26 programs) with a total of 150,068 fault samples of bit flips. The experimental results show the effectiveness of CIVP compared to the state-of-the-art approaches, achieving 20.5%↑ in Recall, and 18.5%↑ in F1-score. Besides, CIVP can more precisely identify vulnerable instructions, sometimes even improving Rec by 114.2% and Pre by 59.5%. Our main contributions are as follows:

- To our knowledge, we are the first to construct a generic model on instruction vulnerability prediction. We propose a novel paradigm for predicting instruction vulnerabilities of unseen programs by using instruction execution context and inherent semantics.
- We utilize LLMs to extract the context of data transfer during instruction execution. By enhancing the instruction's semantics, we can better understand how bit flip errors propagate along data dependencies.
- Our approach captures the whole path of instruction execution to represent the program structure. We also incorporate multi-hop theory into GraphSAGE to ensure that the model can learn inter-program commonalities, which can better represent vulnerable instructions.
- We build a dataset about bit flips containing 26 programs. Extensive experiments with six state-of-the-art baselines are conducted to validate the effectiveness of our approach. We have made our code and data public at https://github.com/SINCOSLab/CIVP/.

II. PRELIMINARIES

Given a program p, which can be compiled into the IR instruction sequence $\chi = \{I_1, I_2, ..., I_N\}$ by *clang*, where N is the total number of instructions. The χ can be divided into a basic block sequence $\varphi = \{B_1, B_2, ..., B_M\}$, where M is the total number of basic blocks. Each basic block B_j consist of a set of instructions$\{I_i | I_i \in B_j, I_i \in \chi\}$. Then, we obtain the control execution, data transfer, function calls, and features, such as opcodes, operands, and registers, during the process of instruction execution. These can be represented as a set of triples $\Gamma = \{t_1, t_2, ..., t_J\}$ and a set of features $\Delta = F_{ins} \bigcup F_{bb}$, where J is the total number of relations, $F_{ins} = \{f_1^I, f_2^I, ..., f_N^I\}$ presents the features of instructions and $F_{bb} = \{f_1^B, f_2^B, ..., f_M^B\}$ presents the features of basic blocks. Each triple t_i is in the form of $(\epsilon_i, r_i, \epsilon_k)$, where $\epsilon_i, \epsilon_k \in \chi \bigcup \varphi$ and r_i is the relation between the entities ϵ_i and ϵ_k. Based on those notations, we can define instruction

[2]https://aosabook.org/en/v1/llvm.html

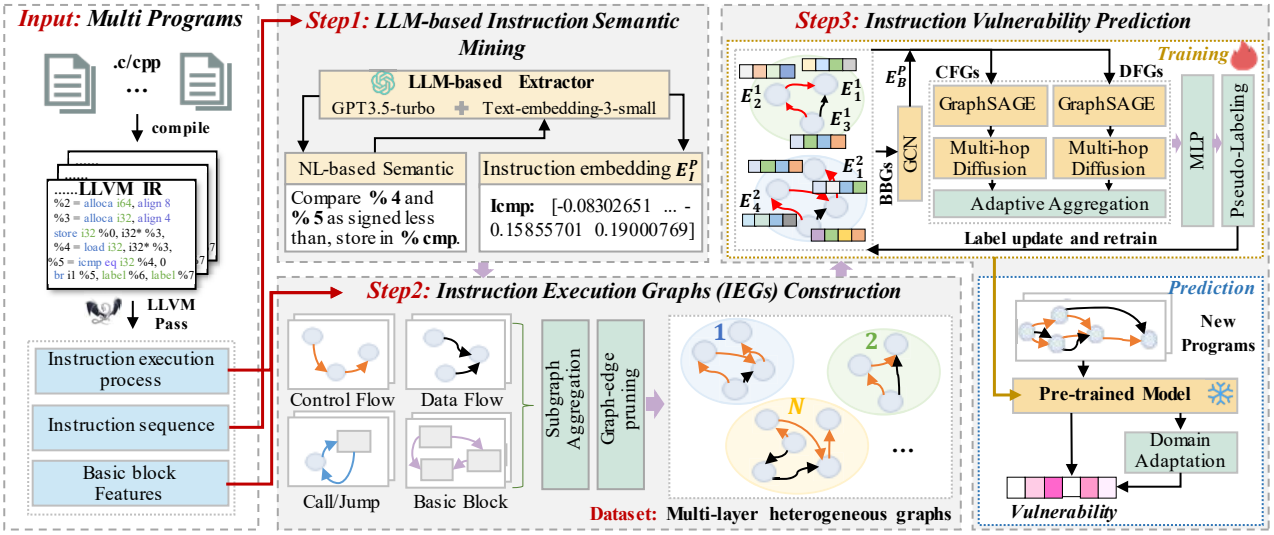


Fig. 3. **The overall framework and some details of CIVP**

vulnerability and formulate the problem of cross-program instruction vulnerability prediction as follows:

Definition 1 (Instruction Vulnerability): Instruction vulnerability is the probability that the program results may be incorrectly raised by the change of instruction I_i due to bit flips during execution, denoted as y_i. Y is the set of all instruction vulnerabilities in the program p.

Problem 1 (Cross-program Instruction Vulnerability Prediction): Given a set of train programs $P = \{p_1, p_2, ..., p_s\}$ and labels $Y_{train} = \{Y_1, Y_2, ..., Y_s\}$, where p_i can be represented as a four-tuple $\{\chi, \varphi, \Delta, \Gamma\}$ and s is the total number of programs. The model training can be formulated as a supervised learning problem:

$$\{p_1, p_2, ..., p_s\} \xrightarrow{\mathcal{F}(\cdot)} Y_{train}. \quad (1)$$

Then, given an unseen program, the instruction vulnerability prediction can be represented as $Y_{new} = \mathcal{F}(\chi, \varphi, \Delta, \Gamma)$.

III. APPROACH

A. Overview

In this section, we introduce the overall architecture of CIVP, an intelligent framework for assisting engineers in discovering instruction vulnerabilities more efficiently before software deployment. As shown in Fig. 3, the framework consists of three main parts: 1) LLM-based instruction semantic mining, aiming to fully represent the inherent semantics of instructions as they are executed. 2) Instruction Execution Graphs (IEG) Construction, using four subgraphs (CFG, DFG, CG, and BBG) to represent the state transfer of instructions during program execution. 3) Instruction Vulnerability Prediction, using graph neural networks and multi-hop diffusion to mine meta-knowledge for better generalization to unseen programs based on IEGs and instruction semantics. The synergy of the above three ensures robustness: LLMs mine deep semantics, GNNs analyze structural dependencies, and multi-hop aggregation identifies transferable rules.

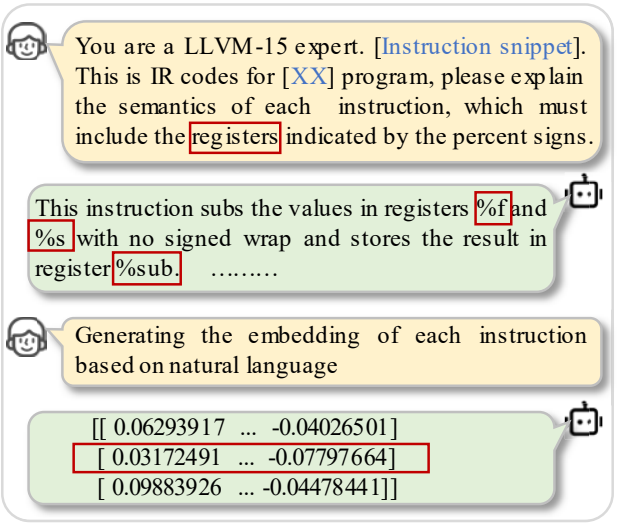


Fig. 4. **The primary prompts for generating NL-based semantics and instruction embeddings (e.g., *"%sub = sub new i32 %f, %s"*).**

B. LLM-based Instruction Semantic Mining

As mentioned above, current approaches solely rely on human-selected heuristic features to represent instructions and lack the inherent semantics of instructions during execution. To fully utilize instruction semantics for exploring instruction correlation during execution, we propose a novel framework called the LLM-based extractor. Fig. 3 (Step 1) depicts the overall framework, which offers a new approach for generating instruction embeddings with implicit data transfer. Given a program p, we compile it to the IR instruction sequence χ. Our approach initially translates the instruction I_i into a natural

language (NL) text Ψ_i^L using prompts. Then, we use a pre-trained text embedding model T_e to generate the instruction semantic embedding E_i^p.

We begin by designing a prompt to parse the IR instruction I_i while preserving the intrinsic semantics of execution. Motivated by the great success of LLM (e.g., ChatGPT [15]) in understanding natural language, we design a specific prompt to guide LLMs in parsing IR instructions, and the primary prompts are shown in Fig. 4. Specifically, we first mine the NL-based semantics of IR instructions from the data transfer perspective during instruction execution. Then, we use a pre-trained model to obtain instruction embeddings according to NL-based semantics. The specific details are as follows:

1) ***NL-based Semantics Generation.*** Although the syntax of IR instructions is simple, they do not fully express their semantics, only reflect the operation. Besides, in software systems, instructions are often communicated during their execution through data transfer (i.e., registers). In case of a bit flip, it may propagate through registers between instructions. Thus, we use LLM to mine NL-based semantics by emphasizing the data transfer process to improve semantic interpretability and reflect instruction intent. As shown in Fig. 4, for instruction text sequence χ, we prompt LLM to mine the NL-based semantics of each instruction (line-by-line) while preserving the registers used by instructions. For example, the instruction *"%sub = sub new i32 %f, %s"* will be translated as *"This instruction subs the values in registers %f and %s with no signed wrap and stores the result in register %sub."*

2) ***Instruction Embedding Generation.*** After obtaining the NL-based semantics, we need to generate the instruction embeddings, which contain the implicit execution process. Instead of using shallow embedding models, we use a smaller LLM (e.g., Text-embedding-3-small) to encode the semantics of text. In particular, given the semantic text N^L, the semantic encoder works as follows:

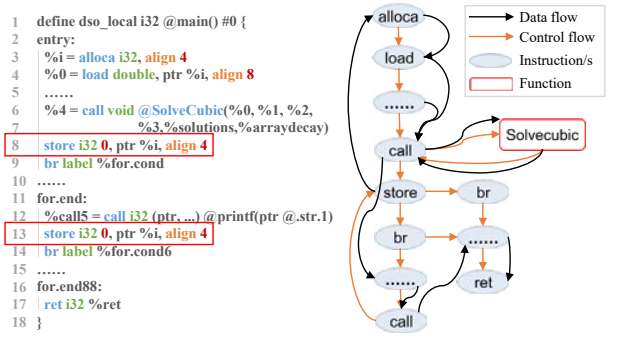
$$E_I^p = T_e(\Psi^L), \tag{2}$$

where $E_I^p \in \mathbb{R}^{N \times d}$ denotes the instruction semantic embeddings, and d is the dimension of the embedding vector. Thus, we can mine the similarity between instructions at the data transfer level to reveal the instruction intent and execution process.

C. Instruction Execution Graphs Construction

To further explore the instruction execution process of programs for mining the propagation pattern of bit flips, we construct the instruction execution graphs based on control and data dependencies, as shown in Fig. 3 (Step 2). The key steps are as follows:

1) ***Subgraph Extraction****:* Through secondary development of the LLVM [16], we develop a program analyzer to extract the process of instruction execution, which contains control flow, data flow, condition jump, function call, and basic blocks, implying error propagation patterns. Then, four types of subgraphs can be obtained. ① Control Flow Graph (CFG): The CFG is a representation, using graph notation, of all paths



(a) IR segments of the Basicmath program

(b) The heterogeneous graph composed of CG, CFG and DFG

Fig. 5. **The graph structures of the Basicmath program**

that might be traversed through a program during its execution [17]. Thus, we extend it to the IR level to extract control dependencies between instructions. Besides, instructions with condition jump relations (e.g., *'icmp'*, *'switch'*, *'invoke'*) are extended to this graph, enhancing the structure semantics of programs. ② Data Flow Graph (DFG): The DFG is a graphical representation of how data moves through a system or software [18]. Thus, we use it to represent data dependencies between instructions during execution at the IR level. ③ Call Graph (CG): The CG graphically represents the call relations between functions in a program. We point the *'call'* instruction to the header of the called function, and the function's *'ret'* instruction points back to the *'call'*. ④ Basic Block Graph (BBG): Some studies have shown that different basic block structures significantly affect the propagation of bit flips [19]. To this end, we construct BBGs to enrich the program representation, exploring the propagation patterns of bit flips better.

2) ***Subgraph Aggregation and Pruning****:* A program frequently contains multiple functions, while the CFG and DFG only represent the control execution and data transfer within a single function. Thus, we add CGs between different DFGs to connect them, as well as CFGs. Besides, we find some instructions with the same intent in CFG or DFG. As shown in Fig. 5 (a), the instruction *"store i32 0, ptr %i, align 4"* has the same intent (storing "0" into register "%i") at different locations. Thus, we drop duplicate instructions and relink the edges, reducing graph complexity and improving training efficiency. Then, we integrate the tuned CFG and DFG to construct a heterogeneous graph as shown in Fig. 5 (b), representing the full execution process of instructions.

3) ***IEG Construction****:* Through the combination of the BBG and heterogeneous graph, we construct an instruction execution graph to represent a program, which is a multi-layer heterogeneous graph, and instructions have edges to the basic blocks they belong to. Then, based on program properties and structure, the instruction features can be expressed as a six-tuple: {*bit widths, predecessors, successors, operands, types, instruction semantics*}. And the basic block features can be expressed as a three-tuple: {*instruction number, predecessors, successors*}. Based on DGL [20], we formally represent the

IEG and incorporate features into the attributes of corresponding nodes. Overall, we build a dataset containing IEGs of 26 programs for training and testing.

D. Instruction Vulnerability Prediction

As shown in Fig. 3 (Step3), the instruction vulnerability prediction model consists of three main components: ① **Basic Block Control Dependency Mining.** As mentioned above, different basic block structures impact the propagation of bit flips differently, so we use the Graph Convolutional Network (GCN) [21] to mine the spatial dependencies of basic blocks, and the generated basic block embeddings will be fused into the initial instruction features. Thus, abnormal jumps can be detected based on unusual control dependencies. ② **Instruction Execution Context Mining.** Then, to mine the different execution contexts of instructions in control dependencies and data transfers, we model CFGs and DFGs separately. We use GraphSAGE [12] and multi-hop theory [22] to summarize the execution context of different programs to better mine the propagation patterns of bit flips with generalization. ③ **Data Augmentation by Pseudo-Labeling.** To solve the scarcity of fault samples and unbalanced label distribution, we use pseudo-labeling [13] to generate more labeled samples, enhancing the discriminative power of the vulnerable instructions.

Let $D_t = \{G^1, G^2, ..., G^s\}$ be an instance of the training set, where s is the total number of programs. Each $G^i = (V^i, E^i)$ denotes the IEG representation of program i, where V^i represents the set of nodes, including node features, and E^i represents the set of edges. In graph neural networks, batch processing is a commonly used technique that aggregates multiple graphs into one large graph for more efficient computation [23]. Thus, based on *"dgl.batch()*[3]*"*, we combine the set of programs D_t into a large graph $L_G =< L_V, L_E >$. Each IEG becomes one disjoint component of the batched graph. The nodes and edges are relabeled to be disjoint segments.

*1) **Basic Block Control Dependency Mining**:* We extract the BBGs: $g_B = (v_B, e_B)$ from L_G, where $v_B \in L_V$ represents the set of basic blocks, and $e_B \in L_E$ represents the set of basic block edges. Then, considering GCN is good at capturing the local patterns in the spatial domain, we apply GCN to mine control dependencies between basic blocks, defined as follows:

$$b_i^l = \sigma(b^{l-1} + \sum_{j \in N(v_B^i)} \frac{1}{\sqrt{|N(j)||N(i)|}} b_j^{l-1} W^{l-1}), \quad (3)$$

where $N(v_B^i)$ represents the neighbor of basic block B_i, l represents the number of GCN layers, and σ represents activation function. The value b^0 of the initial layer is F_{bb}^s. Thus, we can obtain the basic block embeddings: $E_B^s = \{b_1^l, b_2^l, \ldots, b_{M^s}^l\}$, where M^s is the total number of basic blocks in all programs. Then, E_B^s is transmitted to the instruction layer, with each instruction aggregating the embedding of corresponding basic blocks through tensor splicing. Finally, the raw instruction feature F_{ins}^s is updated to $\{I_1^{'}, I_2^{'}, \ldots, I_{N^s}^{'}\}$, where N^s is the total number of instructions in all programs.

[3]https://www.dgl.ai/dgl_docs/en/1.1.x/generated/dgl.batch.html

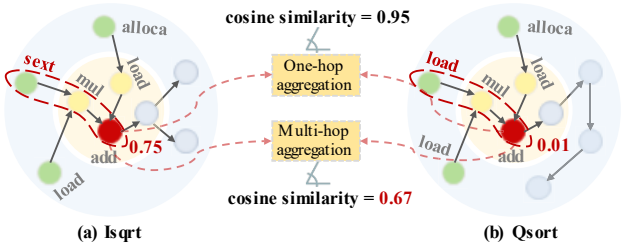


Fig. 6. **Example of multi-hop context for the same instructions across different programs. It shows the context for the same instruction "add" in the Isqrt and Qsort programs. One-hop neighbors present high similarity (0.95), while multi-hop neighbors present lower similarity (0.67).**

*2) **Instruction Execution Context Mining**:* From L_G, the instruction subgraph $g_I = (v_I, e_I)$ can be extracted, where $v_I \in L_V$ represents the set of instructions and $e_I \in L_E$ represents the set of instruction edges. To explore the different propagation patterns of bit flips in control and data flow separately, we divide g_I into the CFGs: $g_I^c = (v_I, e_I^c)$ and DFGs: $g_I^d = (v_I, e_I^d)$. Then, we utilize GraphSAGE and multi-hop diffusion to extract the execution context of each graph, exploring potential propagation patterns of bit flips.

① **CFGs Encoder.** GraphSAGE has shown the capability of generalization to new nodes or graphs by sampling neighbors and aggregating features [24], [25], [26]. Thus, we use GraphSAGE to learn the control dependencies of different programs, mining the key contexts of bit-flip propagation. Using g_I^c and F_{ins}^s as input, a single GraphSAGE layer can be described by the following equation:

$$h_i^l = Relu(W^l \cdot mean(\{h_i^{l-1}\} \cup \{h_j^{l-1}, \forall j \in N(v_I^i)\})), \quad (4)$$

$$h_i^l = h_i^l / ||h_i^l||_2, \quad (5)$$

where h_i^l indicates the instruction embeddings at current state, $N(v_I^i)$ represents the neighbor of instruction I_i and W^l is the learnable weight matrix. After GraphSAGE, we can obtain the instruction embeddings $E_I^c = \{h_1^l, h_2^l, ..., h_{N^s}^l\}$ that contain the context of control dependencies. Then, as shown in Fig. 6, the yellow areas indicate single-hop neighbors, and the blue areas indicate multi-hop neighbors. In program 'Isqrt', the single-hop neighbors of the *add* instruction are *mul* and *load*, and the multi-hop neighbors are *sext*, *load*, and *alloca*. In program 'Qsort', the single-hop neighbors of the *add* instruction are *mul* and *load*, and the multi-hop neighbors are *load*, *load*, and *alloca*. Considering only single-hop neighbors, the embeddings of the two are the same, and it is impossible to distinguish vulnerable instructions, creating noise for critical context analysis. Thus, we use multi-hop diffusion to expand the receptive field of instructions. They can further aggregate the information of local neighbors, improving the mutual heterogeneity among the same instructions. The multi-hop diffusion can be represented as follows:

$$A_i^h = \alpha^{h-1} \frac{1}{|N(v_I^i)|} \sum_{j \in N(v_I^i)} (A_i^{h-1} \cdot A_j^{h-1}) + \beta^{h-1} A^0, \quad (6)$$

where A^0 is the initial input (i.e., $A^0 = E_I^c$), h is the number of hops, and $\alpha, \beta \in \mathbb{R}^h$ are learnable parameters (i.e., $\alpha^h + \beta^h = 1$). It improves the model's performance on unseen programs by mining long-distance dependencies between instructions. The instruction embeddings can be updated to $E_I^C = \{A_1^h, A_2^h, ..., A_{N^s}^h\}$.

② **DFGs Encoder.** To learn the data dependencies of different programs and mine the critical context of data transfer, we utilize the same model as "CFGs Encoder". Using g_I^d and F_{ins}^s as input, we can get the instruction embeddings E_I^d by equation(4-5). Then, we have also found that one-hop neighbors of the same instructions in DFGs have similarities but also exhibit different vulnerabilities. So, we also use multi-hop diffusion to mine long-distance dependencies in DFGs and generate the new instruction embeddings E_I^D.

③ **Pseudo-Labeling.** To measure the impact of control dependencies and data transfer on instruction vulnerability, we apply an adaptive aggregation of E_I^C and E_I^D. Then, through a linear layer, the predicted instruction vulnerabilities Y^p can be obtained, as shown in the following equations:

$$E_I^S = Relu(W^C E_I^C + W^D E_I^D), \tag{7}$$

$$Y^p = softmax(MLP(E_I^S)), \tag{8}$$

And the cross-entropy loss is defined as:

$$\mathcal{L}_s = -\frac{1}{|N_l|} \sum_{i \in D_l} y_i \ln Y_i^p \tag{9}$$

where D_l is the set of labeled instructions, N_l denotes the total number of all labeled instructions in D_l. Finally, we notice that not all instructions need to be executed at program runtime, such as loop branching, so the labels of some instructions are absent. Besides, the imbalanced distribution of vulnerable and non-vulnerable samples across programs leads to data preferences in the model, which affects fairness and overall performance. Pseudo-labeling is a common approach to solve this problem, which applies pseudo-labels to unlabeled samples using the trained model and iteratively repeats this process in the training cycle [27], [28], [29]. To this end, we utilize confidence γ (i.e., hyperparameters) to select pseudo-labels for unlabeled instructions.

$$U_i = \begin{cases} V_u^l & if P(Y_i = 0 | I_i) \geq \gamma, i \notin D_l \\ None & otherwise \end{cases} \tag{10}$$

where V_u^l represents that the instruction is labeled vulnerable, $None$ represents that the instruction is unlabeled. The γ increases gradually as training proceeds, improving generalization ability and model robustness. Then, we retrain the model using the updated label U and calculate the cross-entropy loss:

$$\mathcal{L}_u = -\frac{1}{|N_U|} \sum_{i \in U} U_i \ln \widetilde{Y_i^p} \tag{11}$$

where N_U denotes the total number of all labeled instructions in U, and $\widetilde{Y^p}$ is the newly predicted instruction vulnerabilities. The final loss can be represented as:

$$\mathcal{L}_T = \mathcal{L}_s + \lambda \mathcal{L}_u \tag{12}$$

TABLE I
BASIC INFORMATION OF THE SELECTED PROGRAMS

Dataset	Programs	Static Instructions	Edges	Fault samples
Training Set	Basicmath	201	301	7063
	Isqrt	87	182	4841
	Qsort	211	441	13447
	Dijkstra	319	614	8574
	Fft	252	515	10213
	Float-mm	167	335	9237
	Nbody	430	943	8913
	Factorial	162	335	414
	BinaryTree	222	501	3893
	Fannkuch	432	915	368
	Fasta	463	942	4452
	Mand	231	486	2730
	Bfs	171	439	798
	Schedule	582	1232	9734
Test Set	Rad2reg	55	111	596
	Stack	95	204	717
	Bitstring	94	191	8314
	Datamanager	239	495	2906
	CRC	284	580	1551
	Towers	267	552	26941
	Rot	547	1165	2081
	Replace	1418	3090	3813
	Printtokens	914	1988	3656
	Tcas	287	595	354
	Totinfo	728	1548	4708
	Schedule2	580	1213	9754

In summary, we can obtain a pre-trained model that specializes in instruction vulnerability prediction. For a new program, we use the pre-trained model to directly predict the vulnerabilities of all instructions without retraining. Besides, for programs with poor predictions, we use domain adaptation (i.e., retraining the model with 20% of instructions) to improve the model's performance for the program.

IV. EXPERIMENTAL SETUP

In this section, we evaluate the CIVP and aim to answer the following research questions (RQs):

- **RQ1** : How does CIVP perform compared with the state-of-the-art instruction vulnerability prediction approaches?
- **RQ2**: What is the influence of different components on the prediction performance of our approach?
- **RQ3**: How do different LLMs vary in their ability to generate correct and vulnerability-revealing instruction semantics?
- **RQ4**: How do the different hops impact the performance of CIVP?

A. Datasets

Due to no ground-truth datasets for bit flips, we build the experimental dataset through the following two steps:

① **Program Selection.** To ensure representative and meaningful evaluation, we survey a wide range of prior studies on instruction vulnerability analysis and fault injection (e.g.,

[10], [30], [5], [31]). Based on frequency of use and community consensus, we select 26 representative programs from commonly used benchmarks (e.g., Mibench [32] and Siemens [33]), which appeared most frequently in the literature (as determined by citation count and usage). Table I outlines the basic information of selected programs, in which 'Edges' represents the total number of relations, including control flow, data flow, jump, and so on. All selected programs can be compiled into IR by *clang*, which serves as the input for the fault injection model.

② **Fault Injection.** At present, while some work has investigated the effect of bit flips on program execution, no specialized dataset has been provided. Thus, we employ the LLTFI[4] tool to simulate bit flips. In this paper, we consider transient hardware faults occurring in the processor's computational components, including pipeline stages, flip-flops, and functional units, which interact with registers either directly or indirectly. Thus, we specifically focus on bit flips occurring in the registers manipulated by each instruction. Similarly, faults in processor control logic are considered out of scope due to the presence of internal redundancy or verification mechanisms. Additionally, we ignore faults in instruction encoding, as they are often identified and mitigated through low-level protection techniques such as checksumming or fetch-time validation.

A certain bit of instructions will be flipped each time the fault injection is executed, obtaining the fault pattern of each instruction. Specifically, the fault model M^F can be represented as a five-tuple $\{p, I^T, F^T, N^C, R^e\}$, where p is the program, I^T is the instruction set for fault injection (e.g., *['alloca', 'load', ...]*), F^T is the type of fault (i.e., bit flip), N^C represents the total cycle of fault injections (e.g., number of dynamic instructions), and R^e is the registers for fault injection (e.g., "destination register", "source register"). For each program execution, a simulated error is induced by flipping a bit within the targeted instruction's corresponding register. Finally, the vulnerability of each instruction is calculated by:

$$y_i^p = \frac{Error_i}{B_i \times N_i^C}, \tag{13}$$

where $Error_i$ is the number of errors caused by instruction i after bit flips. B_i is the register bit-width of the instruction i, and N_i^C is the cycle of instruction i. For instance, a *load* instruction has a 64-bit destination register and a 64-bit source register. If it is called 10 times during the program's execution, then the total number of fault injections is $10 * (64 + 64) = 1280$. If program errors occur 500 times among them, then its vulnerability is $500/1280 = 0.39$. Throughout the fault injection, a bit-level inversion is applied to all program instructions, resulting in a total of 150,068 fault samples.

B. Baselines

In this paper, we consider six state-of-the-art (SOTA) approaches to better illustrate the performance difference between SOTAs and CIVP. More precisely, we consider three single-program training approaches (i.e., GATPS [10], PrograML [34] and PerfoGraph [35]), which train the model with partial instructions of the program and then predict vulnerabilities of remaining instructions in the same program. Besides, we also consider three cross-program training approaches (i.e., MVD [36], CPVD [37] and MGVD [38]), where models are trained using multiple programs and applied to unseen programs. We adapt these approaches to suit the instruction vulnerability prediction, as there are currently no relevant approaches available. The details are as follows:

- **GATPS**, which considers four types of instruction edges (i.e., branch, addressing, logical, and define-use) to represent programs and uses graph attention networks to predict error-prone instructions.
- **PrograML**, which captures program control flow, data flow, and call relation between instructions, and is suitable for a wide range of analysis tasks.
- **PerfoGraph**, which is an enhanced compiler and language-agnostic program representation based on PrograML, aggregating data types and numerical information.
- **MVD**, which considers both the program dependence graph and call graph, and uses flow-sensitive graph neural networks to detect vulnerabilities effectively.
- **CPVD**, which uses the code property graph to represent programs and improves the accuracy of vulnerability prediction using domain adaptation.
- **MGVD**, which considers three different forms of structure (i.e., statement texts, raw statement graph, and abstract statement graph) to represent programs and uses the graph neural network and convolutional neural networks to predict vulnerabilities.

C. Implementation Details

Our instruction vulnerability prediction model is implemented in Pytorch-1.10.2 with the Adam optimizer. The learning rate is set to 0.005. $RELU$ is applied as the activation function. GPT-3.5 and Text-Embedding-3-small are used for instruction semantic mining. The dimension of instruction semantic embeddings is set to 128. The hop number for multi-hop diffusion is set to 4. We try our best to reproduce all baselines from publicly available source code and papers and use the same hyperparameter settings as in the original text whenever possible. To ensure the fairness of the experiment, we use the same data split for all approaches. As shown in Table I, we divide the dataset into two sets: first 14 programs for training (80% training, 20% validation), and the remaining for testing. All hyperparameters are tuned based on the performance of the validation set.

D. Evaluation Metrics

We use three basic metrics (Pre, Rec, F1) to evaluate CIVP's performance:

① **Precision (Pre)** = $\frac{TP}{TP+FP}$. The precision measures the percentage of truly vulnerable instructions out of all the

[4]https://github.com/DependableSystemsLab/LLTFI

TABLE II
COMPARISON OF RESULTS FOR INSTRUCTION VULNERABILITY PREDICTION IN DATASETS. THE UNDERLINED PORTION REPRESENTS THE SECOND-BEST OVERALL PERFORMANCE.

Program	GATPS			PrograML			Perfograph			MVD			CPVD			MGVD			CIVP		
	Pre	Rec	F1	Pre	Rec	F1	Pre	Rec	F1	Pre	Rec	F1	Pre	Rec	F1	Pre	Rec	F1	Pre	Rec	F1
Rad2deg	0.80	0.98	0.88	0.79	0.97	0.88	0.81	0.98	0.89	0.87	0.46	0.60	0.82	0.97	0.90	0.87	0.75	0.80	**0.90**	**0.99**	**0.95**
Stack	0.78	0.90	0.84	0.82	0.89	0.86	0.75	0.84	0.79	0.83	0.35	0.51	**0.85**	0.68	0.78	**0.85**	0.56	0.69	**0.85**	**0.99**	**0.91**
Bitstring	0.84	**0.92**	0.86	0.85	0.90	0.86	0.83	0.90	**0.90**	0.89	0.57	0.69	0.83	0.68	0.75	0.86	0.67	0.75	**0.90**	0.84	0.87
Datamanager	0.82	**0.98**	0.90	0.83	**0.98**	0.91	0.86	0.94	0.90	0.84	0.51	0.63	0.86	0.89	0.88	0.81	0.73	0.77	**0.87**	**0.98**	**0.92**
CRC	0.94	0.97	0.97	0.95	0.98	0.97	0.95	**0.99**	0.97	0.95	0.61	0.75	0.97	0.98	0.98	0.96	0.76	0.85	**0.97**	**0.99**	**0.99**
Rot	0.70	**0.82**	0.77	0.75	0.76	0.75	0.74	0.73	0.74	0.75	0.48	0.59	0.86	0.63	0.73	0.81	0.76	0.78	**0.89**	**0.82**	**0.85**
Towers	0.70	0.60	0.64	**0.77**	0.72	0.75	**0.77**	0.67	0.72	0.65	0.48	0.53	0.76	0.50	0.60	0.63	0.75	0.69	**0.77**	**0.82**	**0.79**
Replace	0.50	0.46	0.48	0.65	0.52	0.58	0.38	0.41	0.39	**0.73**	0.31	0.44	0.71	0.50	0.60	0.50	0.48	0.45	**0.73**	**0.91**	**0.81**
Printtoken	0.62	0.69	0.65	0.68	0.75	0.73	0.61	0.59	0.60	0.69	0.33	0.44	0.78	0.80	0.78	0.75	0.38	0.50	**0.80**	**0.81**	**0.80**
Tacs	0.42	0.35	0.38	0.33	0.42	0.27	0.46	0.39	0.42	0.30	0.25	0.28	0.5	0.15	0.24	0.25	0.21	0.26	**0.53**	**0.90**	**0.67**
Totinfo	0.41	0.25	0.31	0.42	0.31	0.33	0.30	0.30	0.30	0.58	0.50	0.54	0.52	0.33	0.40	0.54	0.74	0.63	**0.59**	**0.83**	**0.69**
Schedule2	0.55	0.48	0.51	0.63	0.57	0.59	0.50	0.44	0.47	0.70	0.31	0.43	0.71	0.57	0.63	0.59	0.70	0.64	**0.81**	**0.71**	**0.76**
Average	0.67	0.69	0.68	0.70	0.73	0.70	0.66	0.68	0.67	0.73	0.43	0.53	0.76	0.64	0.69	0.70	0.62	0.65	**0.80**	**0.88**	**0.83**

predicted vulnerable instructions. TP and FP denote the number of true and false positives, respectively.
② **Recall (Rec)** = $\frac{TP}{TP+FN}$. The recall measures the percentage of vulnerable instructions that are predicted out of all vulnerable instructions. FN denotes the number of false negatives.
③ **F1-score (F1)** = $\frac{2*Pre*Rec}{Pre+Rec}$. The $F1$ measures the overall effectiveness by considering both precision and recall.

V. EXPERIMENTAL RESULTS

A. *RQ1: Effectiveness of Vulnerability Prediction*

To answer **RQ1**, we conduct extensive experiments on the test programs for instruction vulnerability prediction. Table II presents the overall performance of each approach concerning the three basic ranking evaluation metrics: Pre, Rec, and $F1$, with the best performances highlighted in bold.
① **Our approach significantly outperforms the state-of-the-art approaches in most programs.** From Table II, we find that CIVP consistently outperforms all baselines across almost all programs (11 out of 12 programs). Specifically, compared to the most competitive baseline, our approach improves 0%-14.1% in Pre, 0%-114.2% in Rec, and 0%-59.5% in $F1$. Additionally, CIVP improves Pre by 5.2%, Rec by 20.5%, and $F1$ by 18.5% on average over the most competitive baseline. This is due to the ability of CIVP to better mine instruction semantics and execution context, removing the ambiguities of instruction context between different programs. However, on the "Bitstring" program, the performance of our approach is a little worse than some single-program training approaches. We find that due to its simple program structure and large fault samples, these approaches may have been able to learn a better representation of instructions. Nevertheless, CIVP also outperforms the cross-program approaches with an increase of 1.1%-8.4% in Pre, 23.5%-47.3% in Rec, and 16.0%-26.1% in $F1$. Overall, CIVP can be used to analyze the instruction vulnerabilities of various programs during the stages of software development and testing.
② **CIVP demonstrates superior performance in identifying vulnerable instructions.** In Table II, we can find that CIVP outperforms all baselines in terms of Pre and has the highest Rec on 11 programs. Additionally, CIVP exhibits excellent generalizability, achieving up to 71% Rec even in the worst-performing "Schedule2" program. It suggests that our approach can better predict truly vulnerable instructions.

Answer to RQ1: CIVP outperforms all state-of-the-art approaches across most programs (11 out of 12 programs) in terms of precision, recall, and F1-score, achieving 5.2%, 20.5%, and 18.5% average improvements. Our approach can more precisely identify vulnerable instructions, sometimes even improving Rec by 114.2% and Pre by 59.5%.

B. *RQ2 Effectiveness of different components in CIVP*

In this section, we explore the impact of different components on the performance of CIVP. Several variants of our approach are introduced as follows. For each variant, we train it from scratch using an equivalent experimental setup while varying individual components. The results are shown in Fig. 7.

- **CIVP-IS**, which removes the instruction semantic mined by LLM, and only uses basic attributes (e.g., bits, types);
- **CIVP-PL**, which removes the Pseudo-Labeling;

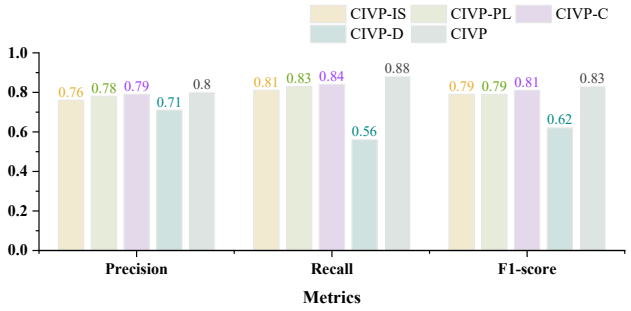


Fig. 7. **Impact of the different components on the performance of CIVP.**

- **CIVP-C**, which removes the CFG of instructions;
- **CIVP-D**, which removes the DFG of instructions.

From Fig. 7, we observe that standard CIVP significantly outperforms all variants in all aspects. The CIVP-D shows the most obvious decrease in predictive effect, with Rec dropping by 36.3%, Pre by 11.2%, and $F1$ by 25.3%, indicating that errors are more likely to propagate with data transfer and affect the execution of programs. **The data dependency between instructions determines the propagation path of bit-flips, significantly influencing the vulnerabilities of instructions.** When bit flips occur in registers or memory, these errors may be loaded into specific instructions and propagate with data transfer, affecting the execution of programs. Besides, the semantics of instructions (CIVP-IS) also have an impact on instruction vulnerability prediction (Rec decreased by 7.9%, Pre by 5%, and $F1$ by 4.8%). It suggests that CIVP can better distinguish between instructions with similar structural positions but differing functionalities or sensitivity to faults by incorporating instruction semantic. Similarly, CIVP-PL and CIVP-C suffer some degradation in performance, by 2.5%-5.6% and 1.2%-4.5%, respectively.

Answer to RQ2: The semantics of instructions and graph structures adequately represent the program execution process. Additionally, Pseudo-Labeling somewhat addresses the imbalance and scarcity of fault samples. Their combined integration enables accurate and robust instruction vulnerability prediction across programs.

C. *RQ3: Performance of different LLMs*

To investigate whether our approach is sensitive to the choice of large language models, we compare the performance of instruction vulnerability prediction using instruction semantics generated by four mainstream LLMs: Qwen-plus, LLaMA-3.1, Deepseek-R1, and ChatGPT-3.5. The results are shown in Fig. 8 and Table III.

As shown, all models achieve comparable performance in terms of precision, recall, and F1-score, with only minor variations (within 1.6%). Among them, Deepseek-R1 yields the best F1-score of 0.841 and Recall of 0.894, followed closely by ChatGPT-3.5 and Qwen-plus. Fig. 8 further reveals that performance trends across 12 different programs

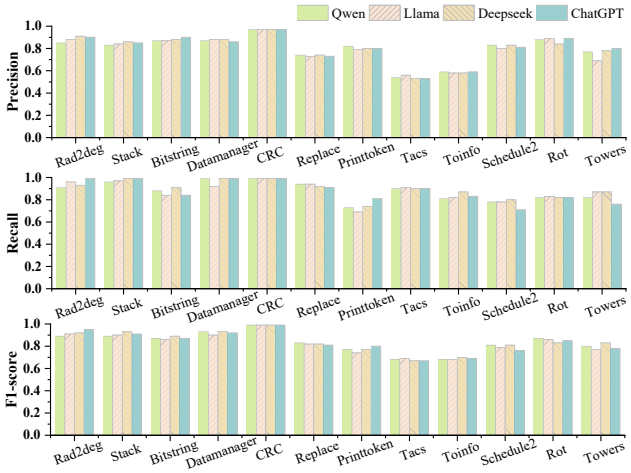


Fig. 8. **Performance of instruction vulnerability prediction at different programs, using the instruction semantics generated by various LLMs.**

TABLE III
PERFORMANCE OF INSTRUCTION VULNERABILITY PREDICTION AT DIFFERENT INSTRUCTION SEMANTICS (AVERAGE ALL PROGRAMS), GENERATED BY VARIOUS LLMS.

LLMs	Precision	Recall	F1-score
Qwen-plus	0.796	0.877	0.832
LLaMa-3.1	0.790	0.876	0.826
Deepseek-R1	0.800	**0.894**	**0.841**
ChatGPT-3.5	**0.801**	0.882	0.834

are largely consistent among models. While minor variations exist for certain benchmarks (e.g., ChatGPT slightly excels on "Rad2deg" and "Printtoken"), the overall rankings remain stable. **This indicates that the quality of generated instruction semantics is generally sufficient across LLMs for effective instruction vulnerability prediction.** Besides, the stability in performance underscores the generalizability of CIVP, attributed to its semantic-enhanced instruction representation and instruction execution context mining. This model-agnostic property is critical for real-world deployment, where LLM availability or computational constraints may vary.

Answer to RQ3: CIVP does not rely on a specific LLM for generating instruction semantics. Even with subtle semantic shifts across different LLMs, our approach maintains consistently high performance. This LLM-agnostic design enhances the practicality and adaptability of our approach for real-world usage.

D. *RQ4 Influences of multi-hop diffusion in CIVP*

To answer **RQ4**, we explore the impact of different hops for instruction vulnerability prediction at all programs. Fig. 9 shows the performance of different hops with three metrics on 6 representative programs, providing a more fine-grained view

of program-specific behaviors. Table IV shows the average performance of all 12 programs.

We observe a consistent trend across most programs: **performance improves as hop increases up to 4**, with Pre increased by 0%-1.2% on average, Rec by 8.6%-23.9%, and $F1$ by 3.7%-10.7%. This confirms that multi-hop diffusion helps capture long semantic dependencies and error propagation paths, which are otherwise missed in most existing approaches. Specifically, it can be seen that the performance dips at 2-hop across several programs (e.g., 'Rad2deg' and 'Stack'), particularly in terms of recall and F1-score. 2-hop neighbors are more likely to include semantically irrelevant or weakly relevant instructions, introducing execution-path ambiguities and corrupting the representation of vulnerable instructions. Then, as the number of hops increases (i.e., CIVP-3 and CIVP-4), instructions across functions or basic blocks help to share semantics for better execution context representation. Interestingly, performance slightly drops or saturates beyond 4 hops (CIVP-5), which suggests that incorporating excessively distant context may introduce noise or dilute the relevance of instruction semantics. This counterintuitive result suggests that directly expanding the neighborhood range does not always lead to better predictions.

Answer to RQ4: In instruction vulnerability prediction, the CIVP's performance can be affected by the hops of diffusion. The 4-hop is probably the best setting for cross-program prediction, which enhances the contrast between instructions by better capturing structural context differences, improving generalization.

E. Threats and Limitations

External Validity. A key threat to validity comes from the dataset we construct. The programs used for evaluation are primarily collected from MiBench and Siemens, while covering a range of application domains such as automotive, industrial control, and network protocols, may not fully represent the diversity and complexity of real-world software systems. The limited number of available programs could constrain the generalizability of CIVP, especially when applied to large-scale software. Furthermore, the real-world deployment environments often involve concurrency, dynamic memory, or low-level optimizations that are not fully reflected in our evaluation setting. In the future, we will consider expanding the dataset with more heterogeneous and representative programs to further enhance the robustness and applicability of our approach.
Internal Validity. Internally, several factors may affect the soundness of our conclusions. First, while we make every effort to fairly reimplement baseline approaches based on their published papers and available code, subtle differences in implementation details or hyperparameter tuning may lead to performance discrepancies. Second, our pseudo-labeling mechanism relies on a threshold to select high-confidence predictions, which may propagate errors or amplify bias if improperly tuned. Finally, our evaluation uses commonly adopted

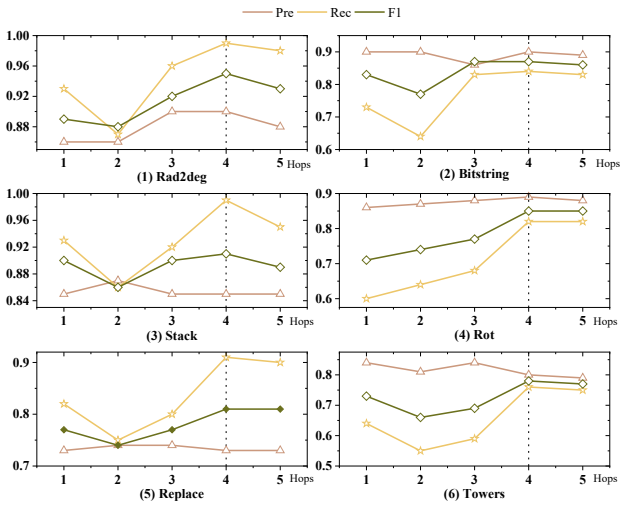


Fig. 9. **The effect of different hops on the performance of CIVP at different programs.**

TABLE IV
COMPARISON OF PRECISION, RECALL AND F1-SCORE FOR INSTRUCTION VULNERABILITY PREDICTION AT DIFFERENT HOPS (AVERAGE ALL 12 PROGRAMS).

Hops	Precision	Recall	F1-score
CIVP-1	0.79	0.79	0.79
CIVP-2	0.80	0.71	0.75
CIVP-3	0.80	0.81	080
CIVP-4	**0.80**	**0.88**	**0.83**
CIVP-5	0.80	0.84	0.81

metrics such as Precision, Recall, and F1-score. However, these do not fully capture the interpretability or robustness of instruction vulnerability prediction in software systems. In the future, we plan to adopt more robust learning techniques and richer evaluation metrics to enhance the credibility of our approach.

VI. Related Work

This section reviews related work from the perspectives of fault injection and instruction vulnerability prediction.

A. Fault Injection

The approach based on fault injection generated fault samples by simulating bit flips and identified vulnerable instructions through statistical analysis [39], [40], [41]. For example, Santos et al. [42] combined the accuracy of Register-Transfer Level (RTL) fault injection with the efficiency of software fault injection, reducing the time from the tens of years an RTL evaluation would need to tens of hours. Then, the most critical GPU resources for bit flips and a set of possible fault effects in instructions could be identified. Li et al. [43] proposed a novel attack for depleting DNN model inference with runtime code fault injections and introduced an LLVM-based automatic vulnerable instruction search algorithm. Ahmad et al. [44] designed an automated framework for selective fault tolerance

of SDCs in software. Then, it performed fault injection to identify the instructions that cause SDCs during bit flips accurately. However, the cost of fault injection increased with program size.

B. Instruction vulnerability prediction

These approaches built a dataset by performing partial fault injection on program instructions to train the model and identify vulnerable instructions, thereby reducing the space-time overhead while ensuring accuracy [45], [30], [46]. Yan et al. [31] proposed an instruction vulnerability prediction model based on a one-class support vector machine classification, which has more accurate vulnerability instruction identification capabilities. However, the selection of features by these approaches was mostly based on the researcher's understanding and could not ensure the effectiveness of these features for downstream tasks. Thus, instruction vulnerability prediction approaches based on program structures and neural networks have received much attention. For example, Jiao et al. [47] exploited the synergy between static analysis and data-driven statistical inference to automatically identify instruction vulnerabilities. By constructing fine-grained bit-level program graphs that encoded error propagation paths, it captured structural factors contributing to instruction susceptibility, thereby enhancing the precision of vulnerability prediction. Yu et al. [48] represented a recent advance in leveraging both instruction information and dynamic fault behaviors for GPU error resilience prediction, and further captured fine-grained bit-level fault propagation using graph neural networks. However, these approaches were still confined to program-specific training, requiring retraining when applied to unseen programs. This limited its scalability and practicality in real-world deployment scenarios where new programs continuously emerged.

In contrast, our work aims to generalize across programs by learning transferable instruction-level patterns of vulnerability, enabling efficient and robust prediction for unseen code without retraining. And our work offers distinct advantages in cost and generalizability compared with non-learning methods. Specifically, ① Cost Efficiency: The fault injection technique requires random or bit-by-bit flipping of each instruction of programs to obtain instruction vulnerabilities. Although it can accurately get instruction vulnerabilities, its time cost increases exponentially with the increase of dynamic instructions (in hours or days). In contrast, CIVP trains a general model offline. For new programs, it can obtain vulnerable instructions relatively accurately without fault injection, greatly reducing the time cost (in seconds). ② Accuracy: Traditional program analysis methods rely on manually defined rules or constraint, their accuracy is difficult to guarantee (e.g., Trident [49] only has an accuracy rate of about 0.58). In contrast, CIVP automatically learns deeper semantic and vulnerability patterns from data, leading to higher accuracy (about 0.81). ③ Generalizability: CIVP can directly infer vulnerable instructions for unseen programs, bypassing the need for expensive dynamic analysis or manual inspection.

VII. Conclusion

In this paper, we propose CIVP, a novel code representation paradigm for efficiently predicting instruction vulnerabilities across programs without retraining. CIVP combines instruction semantics extracted by LLMs with the instruction execution processes (i.e., CFG, DFG, CG, and BBG) to model potential error propagation. Then, we enhance GraphSAGE with multi-hop diffusion to learn transferable instruction-level patterns of vulnerabilities, and use pseudo-labeling to address data scarcity and imbalance. Experiments on 26 programs show that CIVP outperforms state-of-the-art methods. We also release a bit-flip dataset covering 26 programs.

In the future, we plan to ① further expand the dataset with more diverse and large-scale real-world programs, ② develop an open-source tool based on CIVP to assist developers in analyzing and improving the vulnerability resilience of their programs.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China under Grant 62072235.

References

[1] B. Wu, S. Liu, Y. Xiao, Z. Li, J. Sun, and S.-W. Lin, "Learning program semantics for vulnerability detection via vulnerability-specific inter-procedural slicing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1371–1383.

[2] N. Nikiforakis, S. Van Acker, W. Meert, L. Desmet, F. Piessens, and W. Joosen, "Bitsquatting: Exploiting bit-flips for fun, or profit?" in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 989–998.

[3] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1–18.

[4] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

[5] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *2017 47th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2017, pp. 97–108.

[6] B. Nicolescu, Y. Savaria, and R. Velazco, "Software detection mechanisms providing full coverage against single bit-flip faults," *IEEE Transactions on Nuclear science*, vol. 51, no. 6, pp. 3510–3518, 2004.

[7] A. Höller, G. Macher, T. Rauter, J. Iber, and C. Kreiner, "A virtual fault injection framework for reliability-aware software development," in *2015 IEEE International Conference on Dependable Systems and Networks Workshops*. IEEE, 2015, pp. 69–74.

[8] N. Yang and Y. Wang, "Predicting the silent data corruption vulnerability of instructions in programs," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2019, pp. 862–869.

[9] J. Gu, W. Zheng, Y. Zhuang, and Q. Zhang, "Vulnerability analysis of instructions for sdc-causing error detection," *IEEE Access*, vol. 7, pp. 168 885–168 898, 2019.

[10] J. Ma, Z. Duan, and L. Tang, "Gatps: An attention-based graph neural network for predicting sdc-causing instructions," in *2021 IEEE 39th VLSI Test Symposium (VTS)*. IEEE, 2021, pp. 1–7.

[11] M. H. Rahman, S. Di, S. Guo, X. Lu, G. Li, and F. Cappello, "Druto: Upper-bounding silent data corruption vulnerability in gpu applications," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 582–594.

[12] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[13] E. Arazo, D. Ortego, P. Albert, N. E. O'Connor, and K. McGuinness, "Pseudo-labeling and confirmation bias in deep semi-supervised learning," in *2020 International joint conference on neural networks (IJCNN)*. IEEE, 2020, pp. 1–8.

[14] U. K. Agarwal, A. Chan, and K. Pattabiraman, "Lltfi: Framework agnostic fault injection for machine learning applications (tools and artifact track)," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 286–296.

[15] T. B. Brown, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[16] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[17] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 577–587.

[18] J. Xu, Q. Tan, and R. Shen, "The instruction scheduling for soft errors based on data flow analysis," in *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE, 2009, pp. 372–378.

[19] W. Fang, J. Gu, Z. Yan, and Q. Wang, "Sdc error detection by exploring the importance of instruction features," in *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 2021, pp. 351–363.

[20] M. Y. Wang, "Deep graph library: Towards efficient and scalable deep learning on graphs," in *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[21] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[22] G. Wang, R. Ying, J. Huang, and J. Leskovec, "Multi-hop attention graph neural network," *arXiv preprint arXiv:2009.14332*, 2020.

[23] L. Chen, C. Liu, X. Yang, B. Wang, J. Li, and R. Zhou, "Efficient batch processing for multiple keyword queries on graph data," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, pp. 1261–1270.

[24] W. Luo, Y. Zheng, R. Ye, H. Wan, J. Du, P. Liang, and P. Chen, "Satverifiable ltl satisfiability checking via graph representation learning," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1761–1765.

[25] Y. Sun, B. Shi, M. Mao, M. Ma, S. Xia, S. Zhang, and D. Pei, "Art: A unified unsupervised framework for incident management in microservice systems," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1183–1194.

[26] J. Chen, Z. Gong, W. Wang, C. Wang, Z. Xu, J. Lv, X. Li, K. Wu, and W. Liu, "Adversarial caching training: Unsupervised inductive network representation learning on large-scale graphs," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 7079–7090, 2021.

[27] X.-C. Wen, X. Wang, C. Gao, S. Wang, Y. Liu, and Z. Gu, "When less is enough: Positive and unlabeled learning model for vulnerability detection," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 345–357.

[28] Z. Li, M. Pan, Y. Pei, T. Zhang, L. Wang, and X. Li, "Robust learning of deep predictive models from noisy and imbalanced software engineering datasets," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[29] S. Gao, W. Mao, C. Gao, L. Li, X. Hu, X. Xia, and M. R. Lyu, "Learning in the wild: Towards leveraging unlabeled data for effectively tuning pre-trained code models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[30] Q. Lu, G. Li, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Configurable detection of sdc-causing errors in programs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 3, pp. 1–25, 2017.

[31] Z. Yan, Y. Zhuang, W. Zheng, and J. Gu, "Multi-bit data flow error detection method based on sdc vulnerability analysis," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 3, pp. 1–30, 2023.

[32] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.

[33] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, pp. 405–435, 2005.

[34] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. O'Boyle, and H. Leather, "Programl: A graph-based program representation for data flow analysis and compiler optimizations," in *International Conference on Machine Learning*. PMLR, 2021, pp. 2244–2253.

[35] A. TehraniJamsaz, Q. I. Mahmud, L. Chen, N. K. Ahmed, and A. Jannesari, "Perfograph: A numerical aware program graph representation for performance optimization and program analysis," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[36] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "Mvd: memory-related vulnerability detection based on flow-sensitive graph neural networks," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1456–1468.

[37] C. Zhang, B. Liu, Y. Xin, and L. Yao, "Cpvd: Cross project vulnerability detection based on graph attention network and domain adaptation," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4152–4168, 2023.

[38] F. Qiu, Z. Liu, X. Hu, X. Xia, G. Chen, and X. Wang, "Vulnerability detection via multiple-graph-based code representation," *IEEE Transactions on Software Engineering*, 2024.

[39] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "An empirical study of the impact of single and multiple bit-flip errors in programs," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1988–2006, 2020.

[40] X. Xu and M.-L. Li, "Understanding soft error propagation using efficient vulnerability-driven fault injection," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–12.

[41] B. Yuce, N. F. Ghalaty, C. Deshpande, H. Santapuri, C. Patrick, L. Nazhandali, and P. Schaumont, "Analyzing the fault injection sensitivity of secure embedded software," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 4, pp. 1–25, 2017.

[42] F. F. dos Santos, J. E. R. Condia, L. Carro, M. S. Reorda, and P. Rech, "Revealing gpus vulnerabilities by combining register-transfer and software-level fault injection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 292–304.

[43] S. Li, X. Wang, M. Xue, H. Zhu, Z. Zhang, Y. Gao, W. Wu, and X. S. Shen, "Yes,{One-Bit-Flip} matters! universal {DNN} model inference depletion with runtime code fault injection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1315–1330.

[44] H. A.-h. Ahmad and Y. Sedaghat, "An automated framework for selectively tolerating sdc errors based on rigorous instruction-level vulnerability assessment," *Future Generation Computer Systems*, vol. 157, pp. 392–407, 2024.

[45] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 168–179.

[46] Z. Li, H. Menon, K. Mohror, P.-T. Bremer, Y. Livant, and V. Pascucci, "Understanding a program's resiliency through error propagation," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 362–373.

[47] J. Jiao, D. Pal, C. Deng, and Z. Zhang, "Glaive: Graph learning assisted instruction vulnerability estimation," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 82–87.

[48] P. Yu, J. Gu, D. Shen, X. Dong, Y. Liu, and H. Xiong, "Instruction semantics enhanced dual-flow graph model for gpu error resilience prediction," in *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*, 2025, pp. 2803–2814.

[49] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 27–38.