Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection

Xiao Cheng[1], Guanqin Zhang[1], Haoyu Wang[2], Yulei Sui[1]

xiao.cheng@unsw.edu.au

[1]UNSW Sydney, [2]HUST

School of Computer Science and Engineering
UNSW Sydney, Australia

July 26, 2024

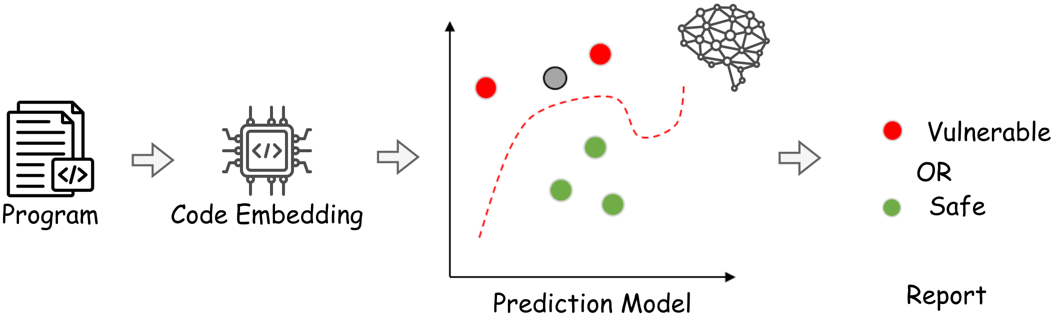
Contribution

- A new path-sensitive code embedding utilizing
  - precise path-sensitive value-flow analysis.
  - a pretrained value-flow path encoder via self-supervised contrastive learning.
- An evaluation to demonstrate the effectiveness and the ability to reduce the training costs of later path-based prediction models to precisely pinpoint vulnerabilities.

Complementary to Static Vulnerability Detection

- Static vulnerability detection has been very successful in detecting low-level, well-defined bugs, such as memory leaks, null dereferences.
- They rely heavily on expert knowledge and user-defined rules.
- They have difficulty in finding a wider range of vulnerabilities (e.g., naming issues and incorrect business logic).

Learning-Based Vulnerability Detection



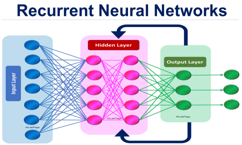
Coarse-grained: predicting whether a program file or a method is safe or vulnerable

Structure-Unaware Embedding

Program

int | main | (| ...

Lexical Tokens



Natural Language Processing

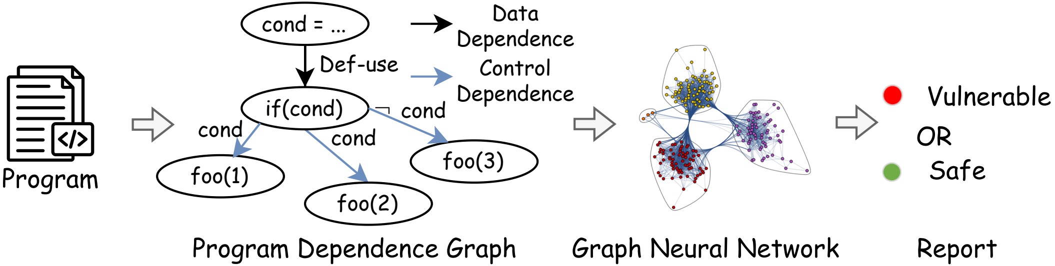
Vulnerable

OR

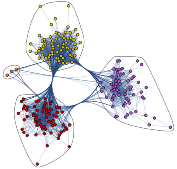
Safe

Report

Structure-Aware Embedding (GNNs)

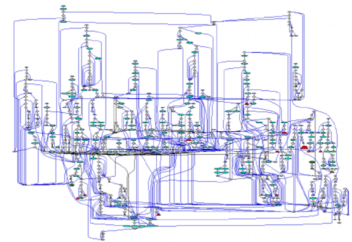


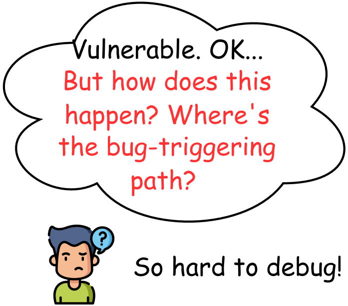
Limitations of GNN (Path-Unaware)



Graph Neural Network

Vulnerable

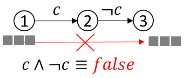


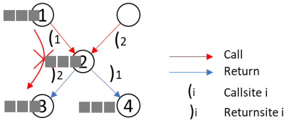


So hard to debug!

Limitations of GNN (Path-Unaware)

GNN does not distinguish feasible/infeasible program dependence paths!





Contradictory path conditions: infeasible

Unmatched call-returns: infeasible

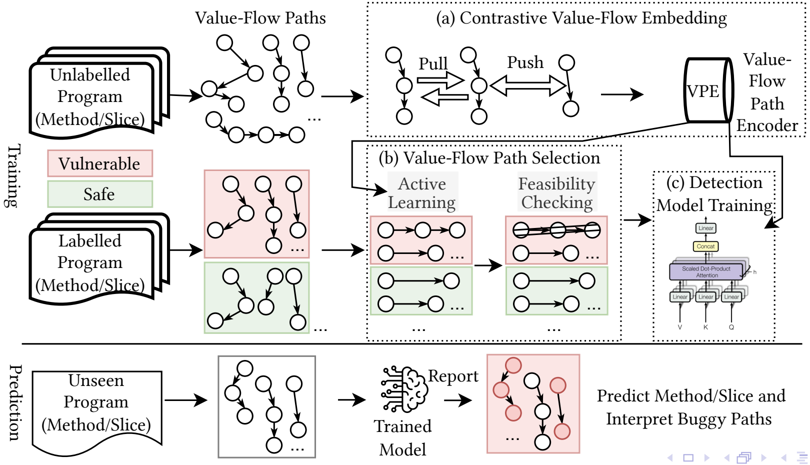
- GNN is **path-unaware** because it uses **all pair-wise message passing**.

$$\mathbf{x}_i' = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \sum_{j \in N(i)} e_{j,i} \cdot \mathbf{x}_j$$

$\mathbf{x}_i$ is the feature vector of node i, $\mathbf{x}_i'$ is the updated feature vector of node i, $N(i)$ is neighbors of node i. $\mathbf{W}_1$, $\mathbf{W}_2$ and $e_{j,i}$ are tunable parameters.

The Aim of This Work

- ContraFlow: a **path-sensitive** code embedding approach which uses self-supervised **contrastive learning** to pinpoint vulnerabilities based on **value-flow paths**.



ContraFlow Framework

(a) Contrastive Value-Flow Embedding

Source Code

```
void msg_q(){
    Inf hd = log_kits("head");
    Inf tl = log_kits("tail");
    ...
    if(FLG){
        rebuild_list(&hd);
        ...
    }else{
        rebuild_list(&tl);
        ...
    }
    if(FLG){
        set_status(&hd,&tl);
    }else{
        log_status(&hd, &tl);
    }
}
```

API misuse: log_kits → rebuild_list → set_status

Can cause unexpected behavior

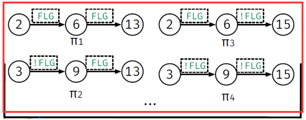
ContraFlow Framework

(a) Contrastive Value-Flow Embedding

Source Code

```
void msg_q(){
    Inf hd = log_kits("head");
    Inf tl = log_kits("tail");
    ...
    if(FLG){
        rebuild_list(&hd);
        ...
    }else{
        rebuild_list(&tl);
        ...
    }
    if(FLG){
        set_status(&hd,&tl);
    }else{
        log_status(&hd, &tl);
    }
}
```

(a) Contrastive Value-Flow Embedding



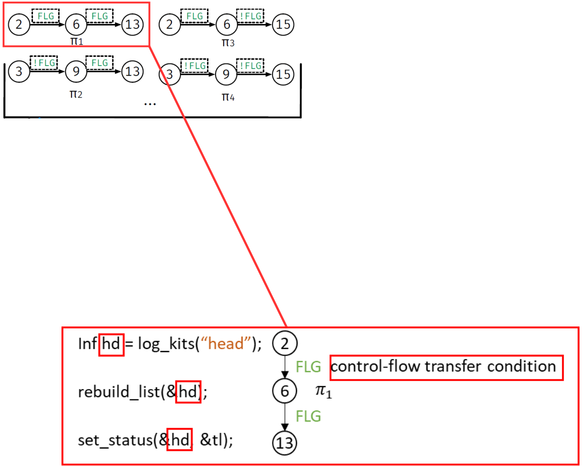
ContraFlow Framework

(a) Contrastive Value-Flow Embedding

Source Code

```
void msg_q(){
    Inf hd = log_kits("head");
    Inf tl = log_kits("tail");
    ...
    if(FLG){
        rebuild_list(&hd);
        ...
    }else{
        rebuild_list(&tl);
        ...
    }
    if(FLG){
        set_status(&hd,&tl);
    }else{
        log_status(&hd, &tl);
    }
}
```

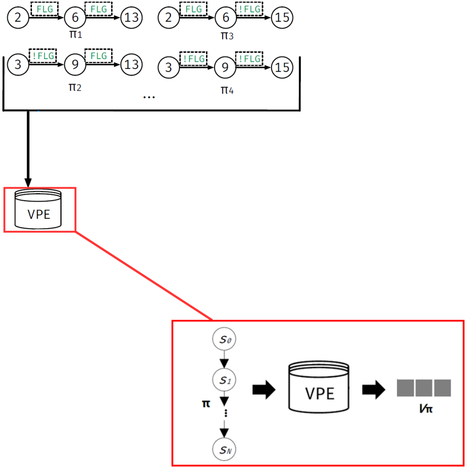
(a) Contrastive Value-Flow Embedding



ContraFlow Framework
(a) Contrastive Value-Flow Embedding

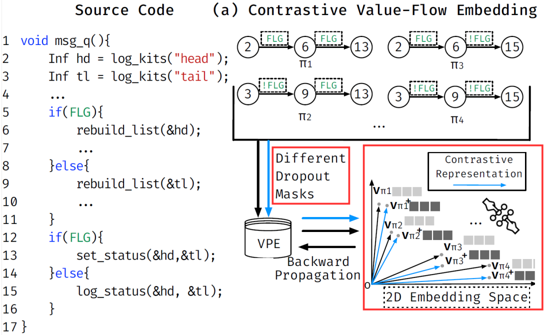
Source Code (a) Contrastive Value-Flow Embedding

```c
void msg_q(){
    Inf hd = log_kits("head");
    Inf tl = log_kits("tail");
    ...
    if(FLG){
        rebuild_list(&hd);
        ...
    }else{
        rebuild_list(&tl);
        ...
    }
    if(FLG){
        set_status(&hd,&tl);
    }else{
        log_status(&hd, &tl);
    }
}
```



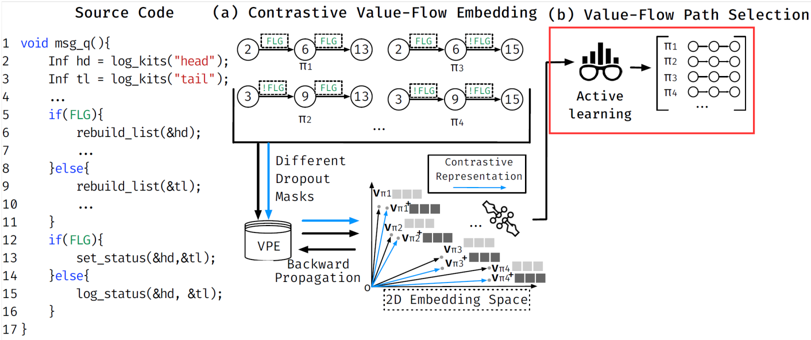
ContraFlow Framework

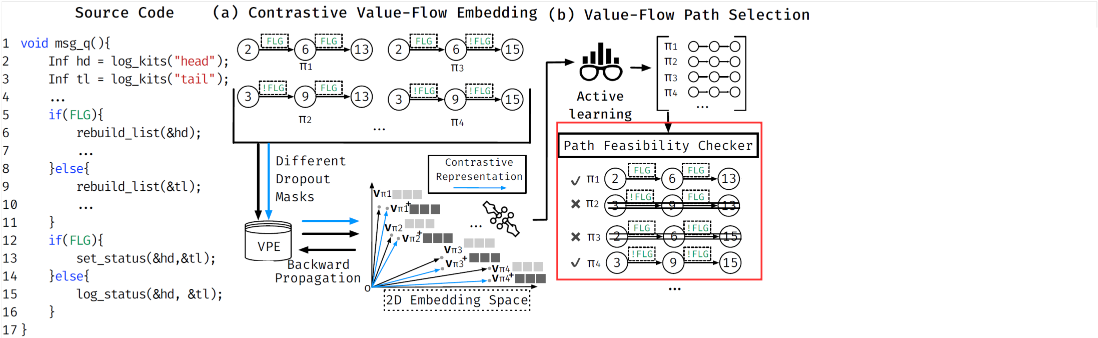
(a) Contrastive Value-Flow Embedding



ContraFlow Framework

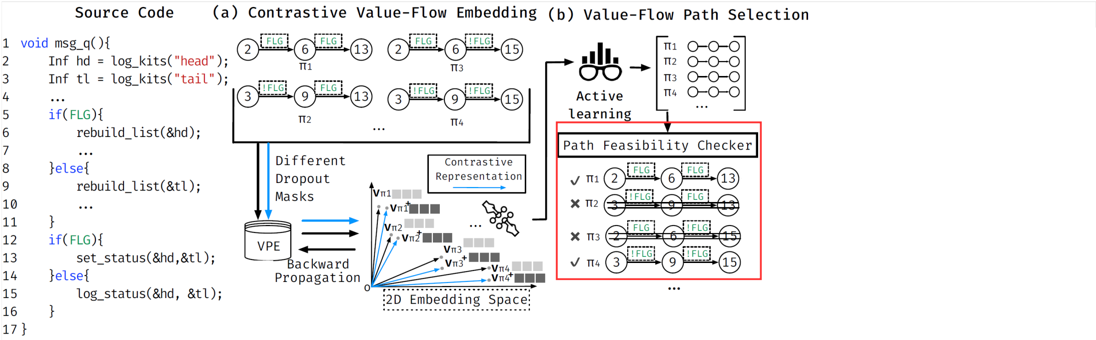
(b) Value-Flow Path Selection





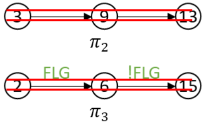
$$guard_v(\pi) = \bigwedge_{i=0}^{N-1} \bigvee_{p \in CP(s_i, s_{i+1})} \bigwedge_{e \in CE(p)} guard_e(e)$$

Value-Flow Guard



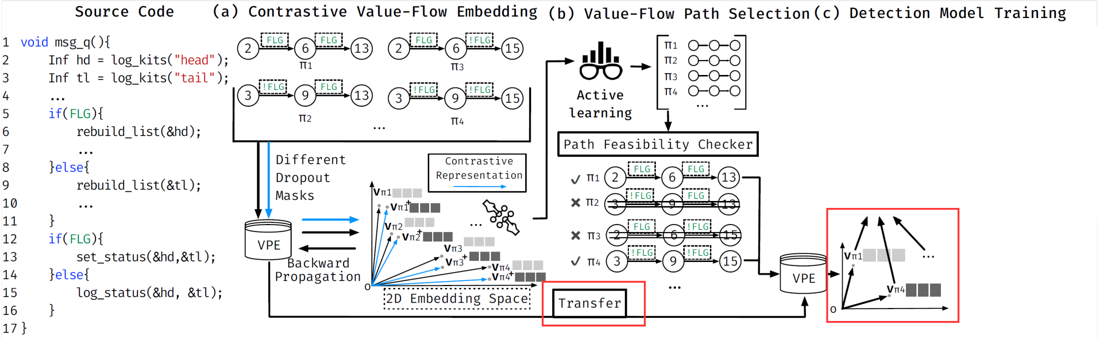
$$guard_v(\pi) = \bigwedge_{i=0}^{N-1} \bigvee_{p \in CP(s_i, s_{i+1})} \bigwedge_{e \in CE(p)} guard_e(e)$$

Value-Flow Guard



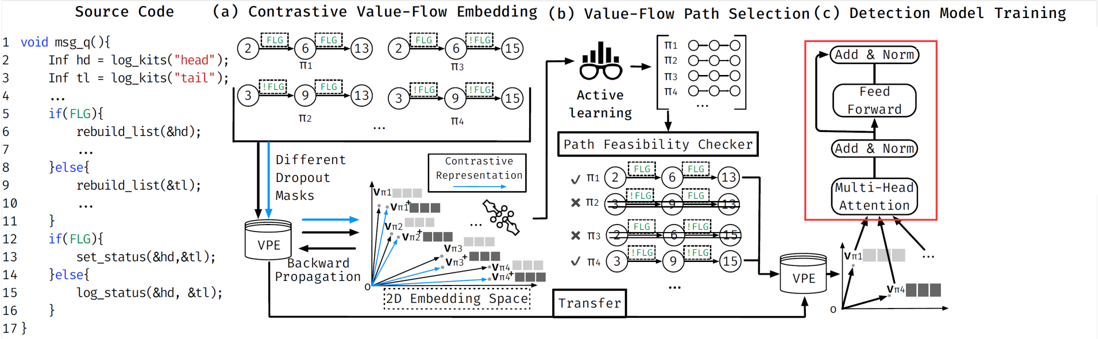
$!FLG \wedge FLG \equiv false$

$FLG \wedge !FLG \equiv false$



Source Code
(a) Contrastive Value-Flow Embedding
(b) Value-Flow Path Selection
(c) Detection Model Training
1 void msg_q(){
2 Inf hd = log_kits("head");
3 Inf tl = log_kits("tail");
4 ...
5 if(FLG){
6 rebuild_list(&hd);
7 ...
8 }else{
9 rebuild_list(&tl);
10 ...
11 }
12 if(FLG){
13 set_status(&hd,&tl);
14 }else{
15 log_status(&hd, &tl);
16 }
17 }
FLG
!FLG
π1
π2
π3
π4
Different Dropout Masks
VPE
Backward Propagation
Contrastive Representation
2D Embedding Space
Active learning
Path Feasibility Checker
Transfer

ContraFlow Framework
(c) Detection Model Training

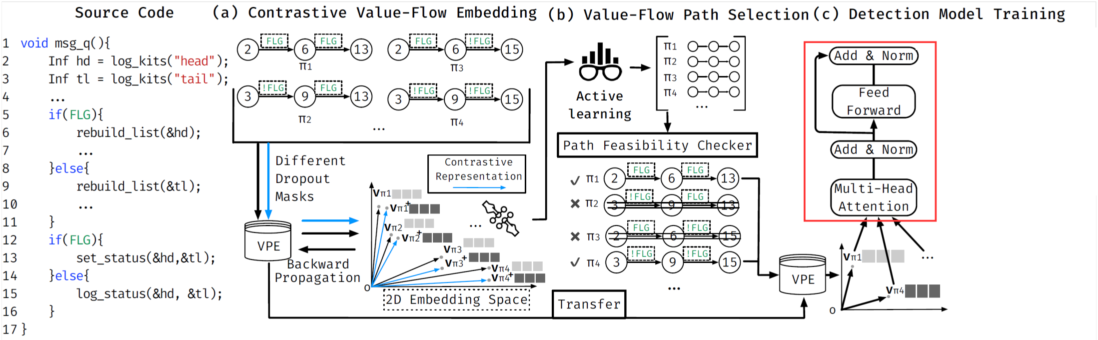


$$\mathbf{V}' = [\mathbf{h}_1 || ... || \mathbf{h}_h]\mathbf{W}^o$$

$$\mathbf{h}_i = Attn(\mathbf{V}\mathbf{W}_i^Q, \mathbf{V}\mathbf{W}_i^K)(\mathbf{V}\mathbf{W}_i^V)$$

$$Attn(\mathbf{Q}, \mathbf{K}) = softmax(norm(\mathbf{Q}\mathbf{K}^\top))$$

Multi-head self-attention

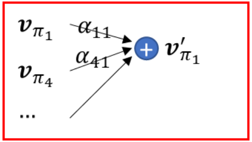


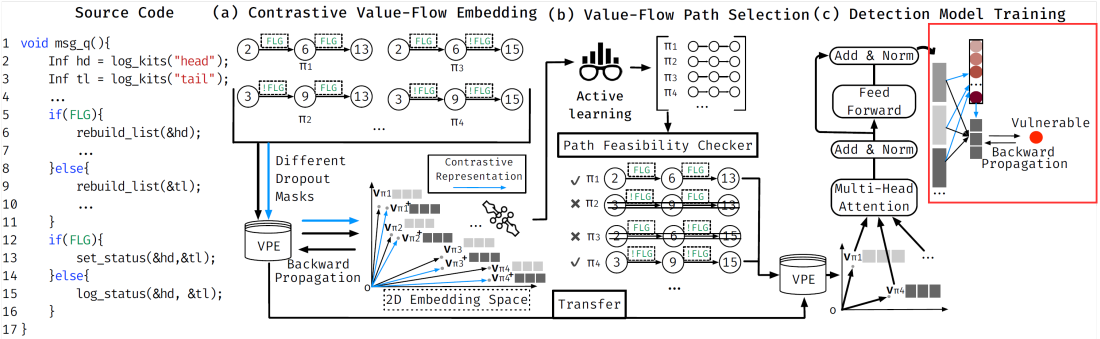
$$\mathbf{V}' = [\mathbf{h}_1 || ... || \mathbf{h}_h]\mathbf{W}^o$$

$$\mathbf{h}_i = Attn(\mathbf{V}\mathbf{W}_i^Q, \mathbf{V}\mathbf{W}_i^K)(\mathbf{V}\mathbf{W}_i^V)$$

$$Attn(\mathbf{Q}, \mathbf{K}) = softmax(norm(\mathbf{Q}\mathbf{K}^\top))$$

Multi-head self-attention

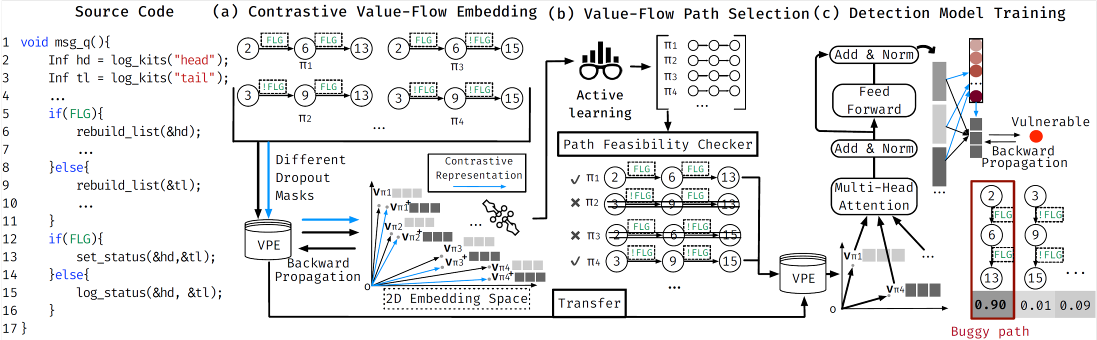




$$\alpha_i^c = \frac{exp(\mathbf{v}_{\pi_i}^\top \mathbf{a}_c)}{\sum_{j=1}^N exp(\mathbf{v}_{\pi_j}^\top \mathbf{a}_c)}$$

$$\mathbf{v}_c = \sum_{i=1}^N \alpha_i^c \cdot \mathbf{v}_{\pi_i}$$

soft attention



highest attention weights!

OpenSSL Cryptography and SSL/TLS Toolkit
APACHE HTTP SERVER PROJECT
libav
FFmpeg
NGINX
TIFF
CVE® Common Vulnerabilities and Exposures
QEMU
......

288 open-sourced projects
30Million lines of code
275K programs

BUFFER_OVERRUN_L1
BUFFER_OVERRUN_L2
BUFFER_OVERRUN_L3
BUFFER_OVERRUN_S2
INTEGER_OVERFLOW_L1
INTEGER_OVERFLOW_L2
INTEGER_OVERFLOW_R2
MEMORY_LEAK
NULL_DEREFERENCE
RESOURCE_LEAK
UNINITIALIZED_VALUE
USE_AFTER_FREE
......

[7] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI Based Vulnerability Detection Methods Using Differential Analysis. In Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). ACM, New York, NY, USA.
[8] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In Proceedings of the 17th International Conference on Mining Software Repositories (MSR). ACM, 508–512. https://doi.org/10.1145/3379597.3387501
[9] YaQin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS '19). Curran Associates Inc. https://doi.org/10.5555/3454287.3455202

Table 1: Labeled sample Distribution.

Dataset	granularity	# Vulnerable	# Safe	# Total
D2A	Method	21,396	2,194,592	2,215,988
	Slice	105,973	10,983,992	11,089,965
Fan	Method	8,456	142,853	151,309
	Slice	42,527	713,239	717,496
FQ	Method	8,923	9,845	18,768
	Slice	45,627	50,125	95,752
Total	Method	38,775	2,347,290	2,386,065
	Slice	194,127	11,747,356	11,903,213

Table 2: Comparison of method- and slice-level approaches under informedness (IF), markedness (MK), F1 Score (F1), Precision (P) and Recall (R). ContraFlow-method/slice denotes the evaluation at method- and slice-level respectively.

Model Name	IF (%)	MK (%)	F1 (%)	P (%)	R (%)
VGDetector	31.1	29.3	56.7	52.6	61.4
Devign	30.1	28.8	58.7	54.6	63.4
Reveal	34.2	33.8	63.4	61.5	65.5
ContraFlow-method	**60.3**	**58.2**	**75.3**	**71.5**	**79.4**
VulDeePecker	17.3	17.3	52.3	52.2	52.4
SySeVR	24.3	24.2	55.0	54.5	55.4
DeepWukong	48.1	48.4	67.0	67.4	66.5
VulDeeLocator	38.4	38.1	62.0	61.4	62.5
IVDetect	37.4	37.3	64.1	64.0	64.6
ContraFlow-slice	**75.1**	**72.3**	**82.8**	**79.5**	**86.4**

Future Research Opportunities

- Pushing the boundaries to scale more precise software security analysis (on-demand, selective, and adaptive) for detecting emerging vulnerabilities.
- **SA4AI**: **Abstract execution to analyse and verify code LLMs / neural networks**.
  - Symbolic path-sensitive analysis to prove properties of neural networks, such as robustness, safety, and security guarantees of code LLMs, as well as understanding and explanation.
- **AI4SA**: **Ultra-fast learning-based vulnerability detection** to significantly boost the performance of conventional software analysis
  - Robust, comprehensive learning-based code representations with deep code semantics (e.g., path-sensitive abstractions).

Thank You!
Q & A