CE 815 – Secure Software Systems

ML-Based Vulnerability Detection Methods (Vulchecker)

Mohammad Haddadian/Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. A reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.

Review

- Automated vulnerability detection
- Code graph representation
- Word2Vec
- GNN
- Hand-selected dataset

- Problem?

Prior Works Limitations

- Detects vulnerability at function level
- Can’t find vulnerability type

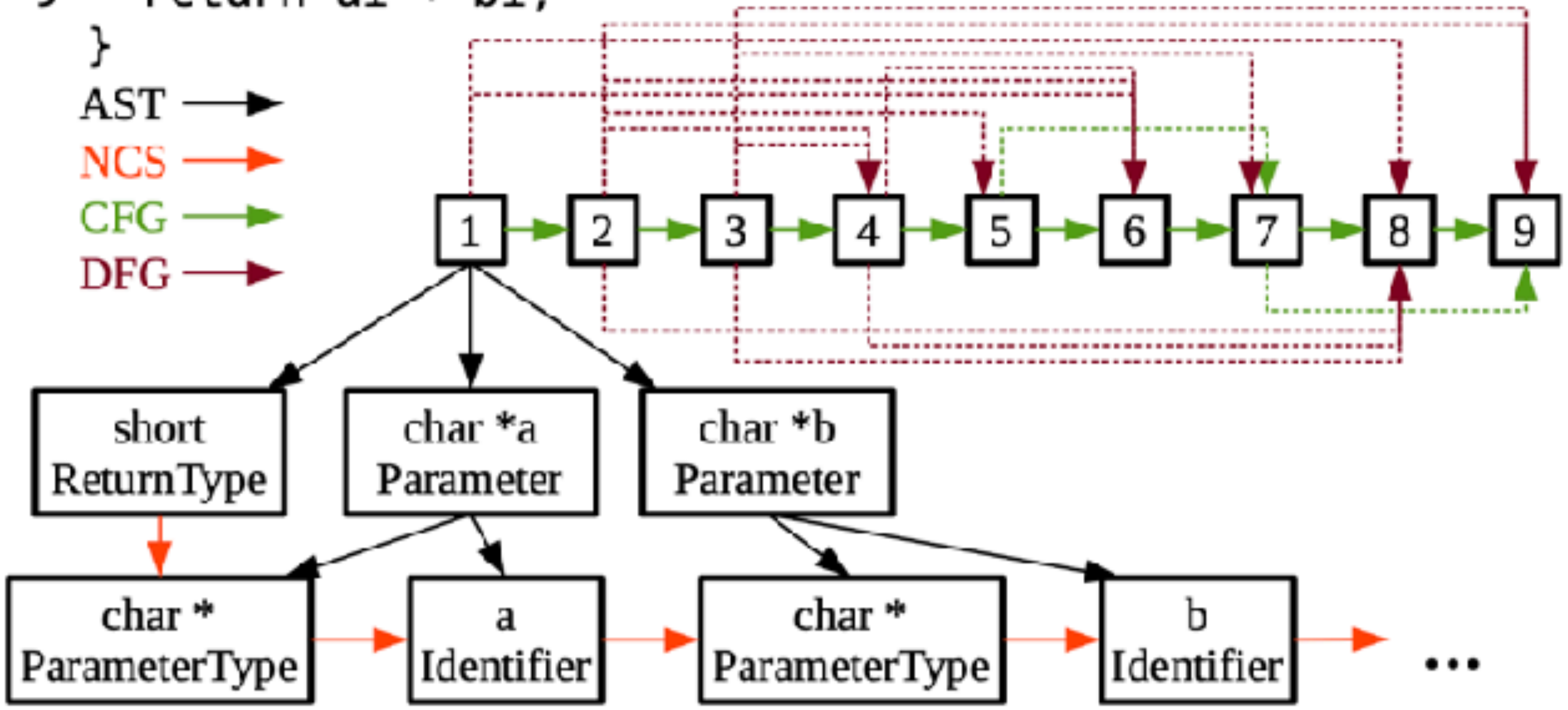
VulChecker

- Precisely locate vulnerabilities in source code (down to the exact instruction)
- Classify vulnerabilities type
- Low-cost dataset augmentation

Insights

- Broad Program Slicing
  - Location of the vulnerability instead of a region or function
- Incomplete Code Representations
  - enhanced-PDG
- Manifestation distance
  - Manifestation vs root cause
- The Lack of Labeled Data
  - datasets that only label code regions or functions
- Level of program representation
  - Source code or machine instructions

```c
short concat(char *a, char *b, char **out) {
  short al = strlen(a);
  short bl = strlen(b);
  *out = (char *) malloc(al+bl);
  if (al)
    memcpy(*out, a, al);
  if (bl)
    memcpy(*out+al,b,bl);
  return al + bl;
}
```



Prior Works

			(1) Code Representation								(2) Sample Selection							(3) Feature Extraction					(4) Model induction				(5) Application				
			Level		Structure						Code Slicing: Plane			Pol		Cut		Node				Edge	Input		Model	Utilizes Edge Type	Detection level				Classifies Vuln. Type?
Year	Cite	Name	Source Code	IR	Linear	CFG	PDG	CPG	ncsCPG	ePDG	Function	Control-flow	Data-flow	Generic	Manifestation	Region	Scoped	One-hot Enc.	Word2Vec	Doc2Vec	Explicit features	Dtype feature	Sequence	Graph			Function	Code Region	Line	Instruction	
2018	[28]	Russle'18	•		•						•					•		•					•		CNN,RF		•				•
2018	[23]	Vuldeepecker	•		•								•	•		•		•					•		BiLSTM			•			•
2019	[40]	μVulDeePecker	•		•								•	•		•		•					•		BiLSTM			•			•
2019	[39]	Devign	•						•		•					•			•					•	GCN,DNN	•	•				
2019	[14]	VGDetector	•			•					•					•				•				•	GCN,DNN			•			
2019	[31]	NW-LCS	•			•					•					•		•						•	LCS Scores		•				•
2020	[19]	Li'20	•		•							•		•		•			•				•		CNN			•			•
2020	[38]	Zagane'20	•		•							•		•		•		○					•		DNN			•			
2020	[32]	Funded	•						•		•					•			•					•	GNN,GRU	•	•				•
2020	[30]	AI4VA	•					•			•			•		•			•					•	GNN,GRU	•	•				
2021	[22]	SySeVR	•		•							•		•		•		•					•		BiRNN			•			
2021	[20]	Li'21		•	•							•		•		•		•					•		CNN+RNN,DNN			•			
2021	[21]	Vuldeelocator		•	•							•			•	•		•					•		BiRNN				•		
2021	[13]	DeepWukong	•				•					•		•		•				•				•	GCN,DNN			•			•
2021	[35]	Wu'21	•				•				•					•			•					•	GNN,DNN	•	•				
2021	[9]	BGNN4VD	•					•			•			•		•			•					•	GNN,GRU	•	•				
2021	[11]	Reveal	•					•			•			•		•			•					•	GCN,DNN	•	•				
		VulChecker		•						•		•			•		•	○			•	•		•	GN (S2V)	•			•	•	•

VulChecker

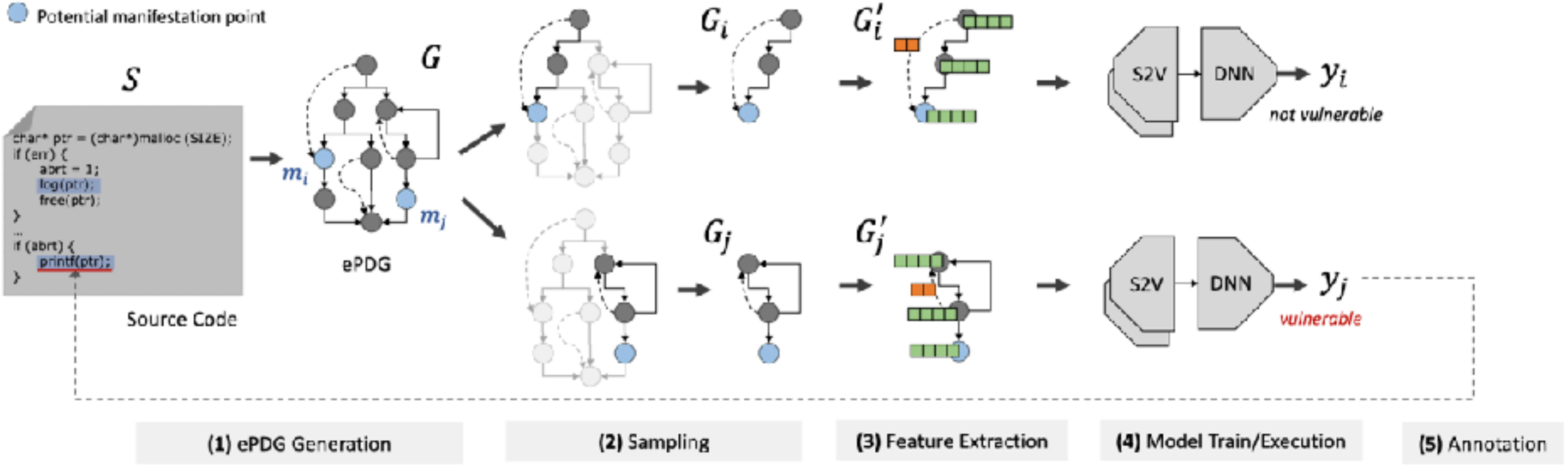
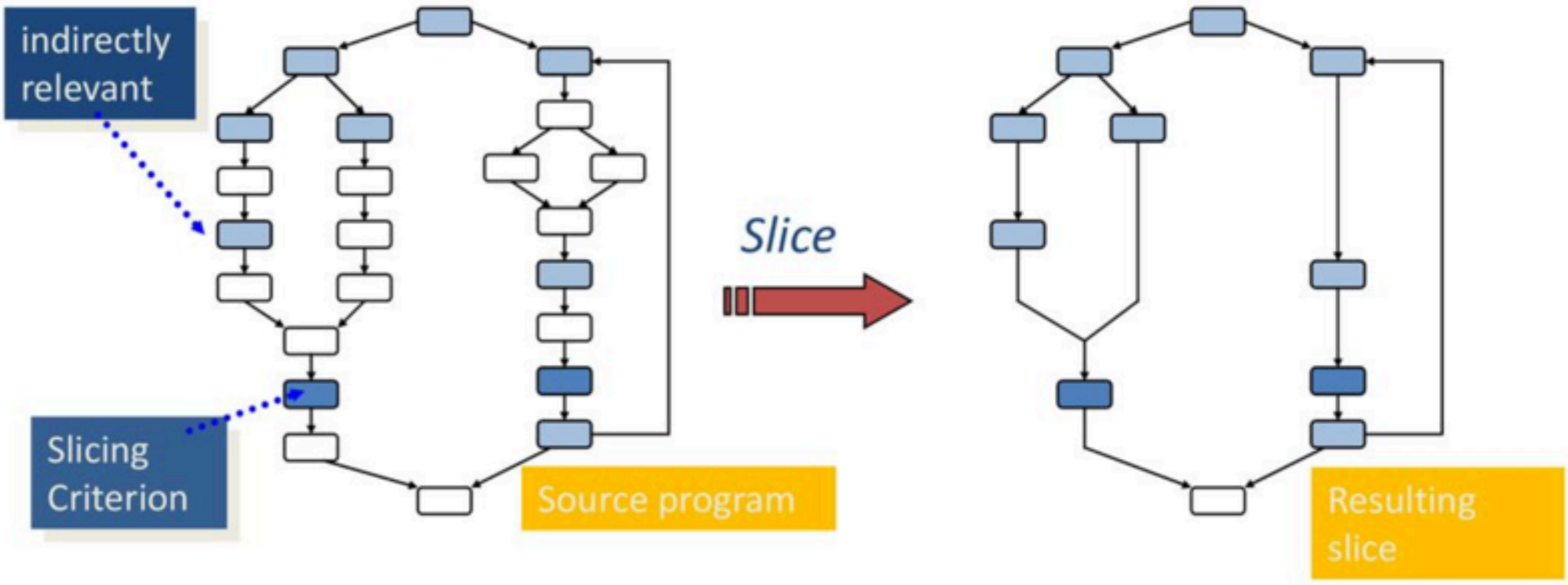


Figure 2: A diagram showing the steps of VulChecker's pipeline for one CWE. Note that the real graphs are significantly larger than what is visualized (e.g., projects like `libgit2-v0.26.1` have over 18 million nodes in G). Solid edges represent control-flow and dashed edges are data dependencies.

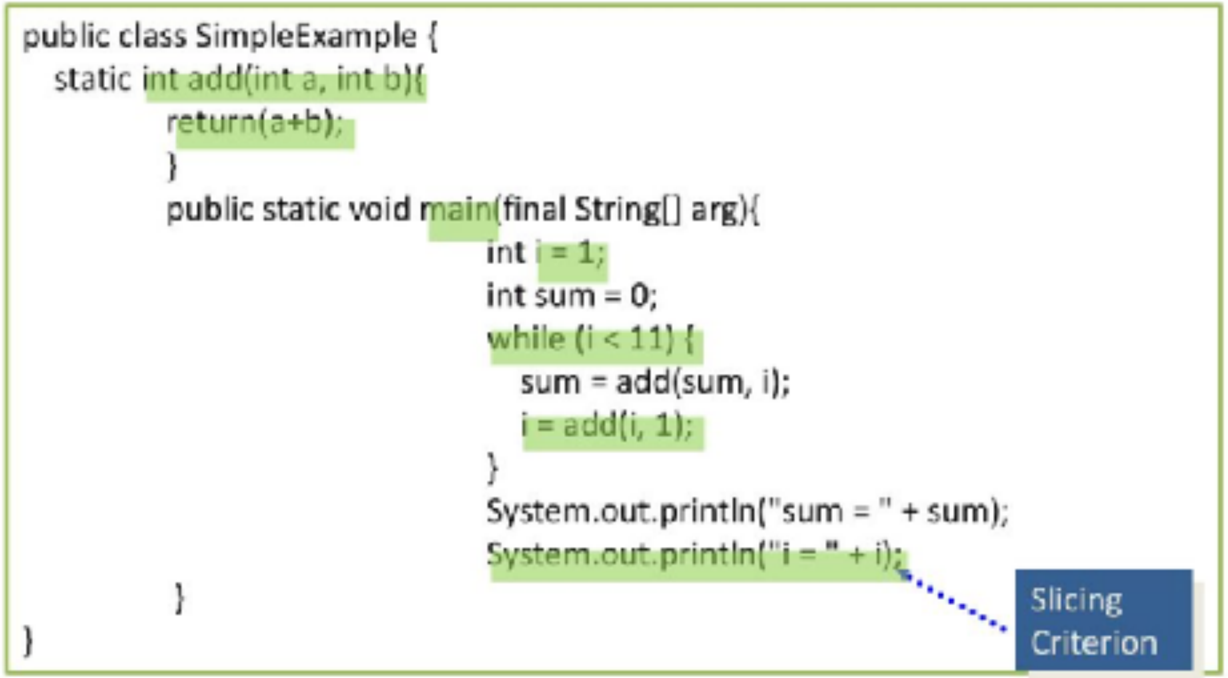
ePDG

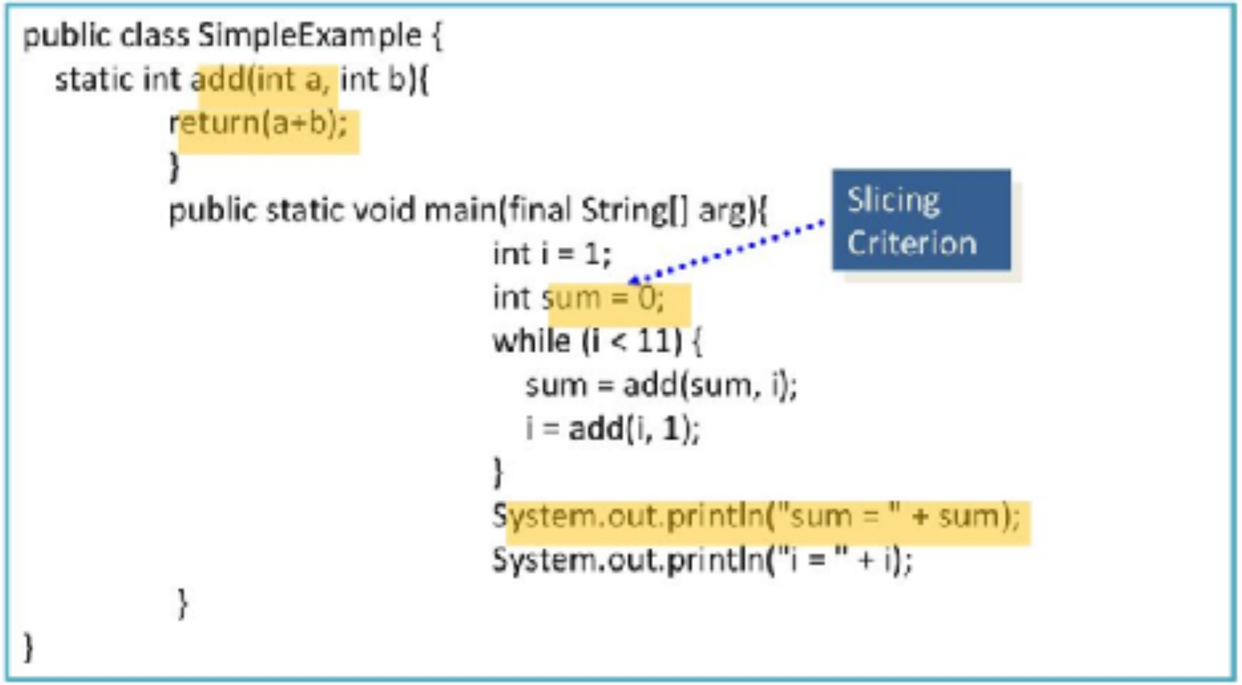
- ePDGs are graph structures in which nodes represent atomic machine-level instructions and edges represent control- and data-flow dependencies between instructions

Program Slicing



Program Slicing (cont.)





VulChecker

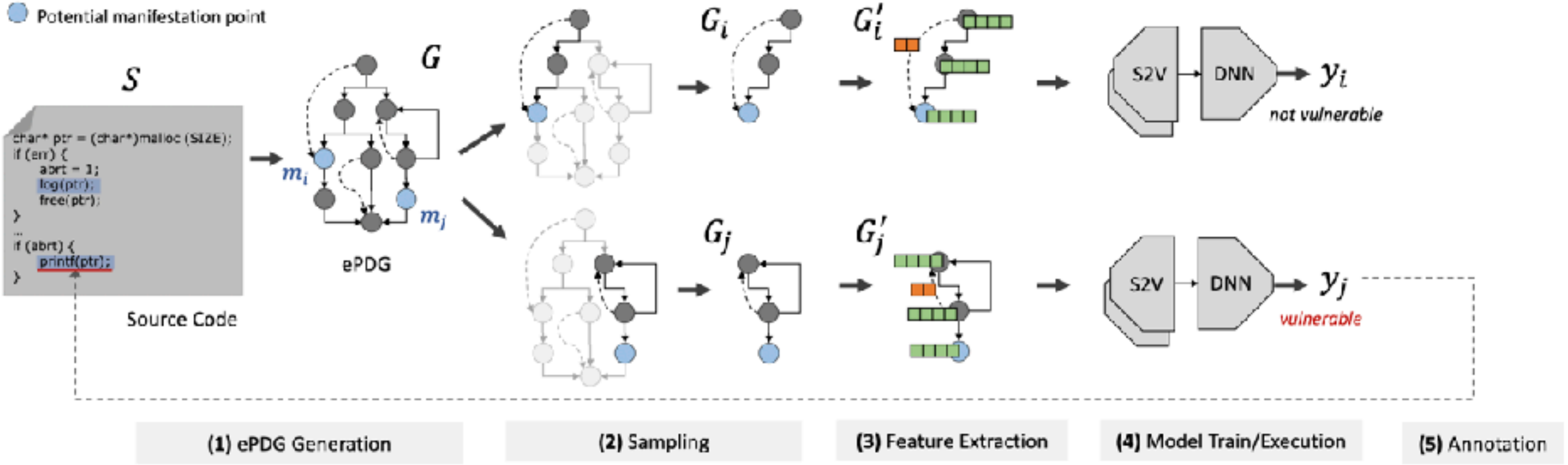


Figure 2: A diagram showing the steps of VulChecker's pipeline for one CWE. Note that the real graphs are significantly larger than what is visualized (e.g., projects like `libgit2-v0.26.1` have over 18 million nodes in G). Solid edges represent control-flow and dashed edges are data dependencies.

ePDG Generation

- Lowering the source code S to LLVM IR
- Extracting G based on the structure and flows it contains

Lowering Code to LLVM IR

- Simplifies the program representation:
  - Control-flow: complicated branching constructs in source code are reduced to conditional jumps that test a single condition
  - Data-flow: definition-use chains are shorter and less complex as they are based on virtual register values rather than source code variables
- During lowering, VulChecker instructs Clang to embed debug information in the IR, which enables traceability of IR instructions back to source code instructions

Lowering Code to LLVM IR (cont.)

- Using semantic-preserving compiler optimizations provided by LLVM to simplify and better express the code in *G:*
  - Function inlining to replace function call sites in the IR with a concrete copy of the called function body
  - Indirect branch expansion to eliminate indirect branching constructs
  - Dead code elimination to reduce the size of the output graph

Generating the ePDG

- C is the set of all types of instructions in the LLVM instruction API (e.g., return, add, allocate, etc.) and A_c is the set of all possible attributes for instruction $v \in V$ of type c.
- D is the set of edge types (i.e., control-flow or data-flow) and A_d is the set of flow attributes for a flow type d (e.g., the data type of the data dependency)

$$G := (\mathcal{V}, \mathcal{E}, q, r)$$

$$q : \mathcal{V} \rightarrow \{\{c, a\} : c \in C, a \in A_c\}$$

$$r : \mathcal{E} \rightarrow \{\{(x, y), d, b\} : x, y \in \mathcal{V}, d \in D, b \in A_d\}$$

Sampling

- PoI Criteria
- Program Slicing
  - Crawls G backwards from m_i using breadth first search (BFS)
- Labeling

Feature Extraction

- Operational Node Features
- Structural Node Features
  - Distance from the nearest potential root cause
  - Betweeness centrality measure (BEC)
- Semantic Node Features
- Edge Features

Table 2: Summary of Features used in G_i'

	Name	Type: Bool	Type: Num.	Type: Categ.	Count
Vertex	Has static value?	•			1
Vertex	Static value		•		1
Vertex	Operation {+, *, %, ...}			•	54
Vertex	Basic function {malloc, read, ...}			•	1228
Vertex	Part of IF clause	•			1
Vertex	Number of data dependents		•		1
Vertex	Number of control dependents		•		1
Vertex	Betweeness centrality measure		•		1
Vertex	Distance to m_i		•		1
Vertex	Distance to nearest r		•		1
Vertex	Operation of nearest r			•	54
Vertex	Output dtype {int, float, ...}			•	6
Vertex	Node tag {r, m, none}	•			2
Vertex				**Total**	1352
Edge	Output dtype {float, pointer ...}				6
Edge	Edge type {CFG, DFG}				2
Edge				**Total**	8

Embedding

- Some embeddings include one hot encodings and pre-processed embeddings (e.g., Word2Vec)
- In some cases entire portions of code are summarized using Doc2Vec
- The issue with these representations:
  - nodes in G_i would likely capture multiple operations in a single line of source code resulting in a loss in semantic precision
  - the use of pre-processed embeddings prevents the model from learning the best representation to optimize the learning objective

Data Augmentation

- Data augmentation is a technique for creating new training examples from existing ones. VulChecker augments its training dataset by adding synthetic vulnerabilities to "clean" projects.
- Validity: Since augmentation process splices multiple ePDGs, it may produce samples where a vulnerability ePDG subgraph lies on an infeasible path in the augmented ePDG

Data Augmentation (cont.)

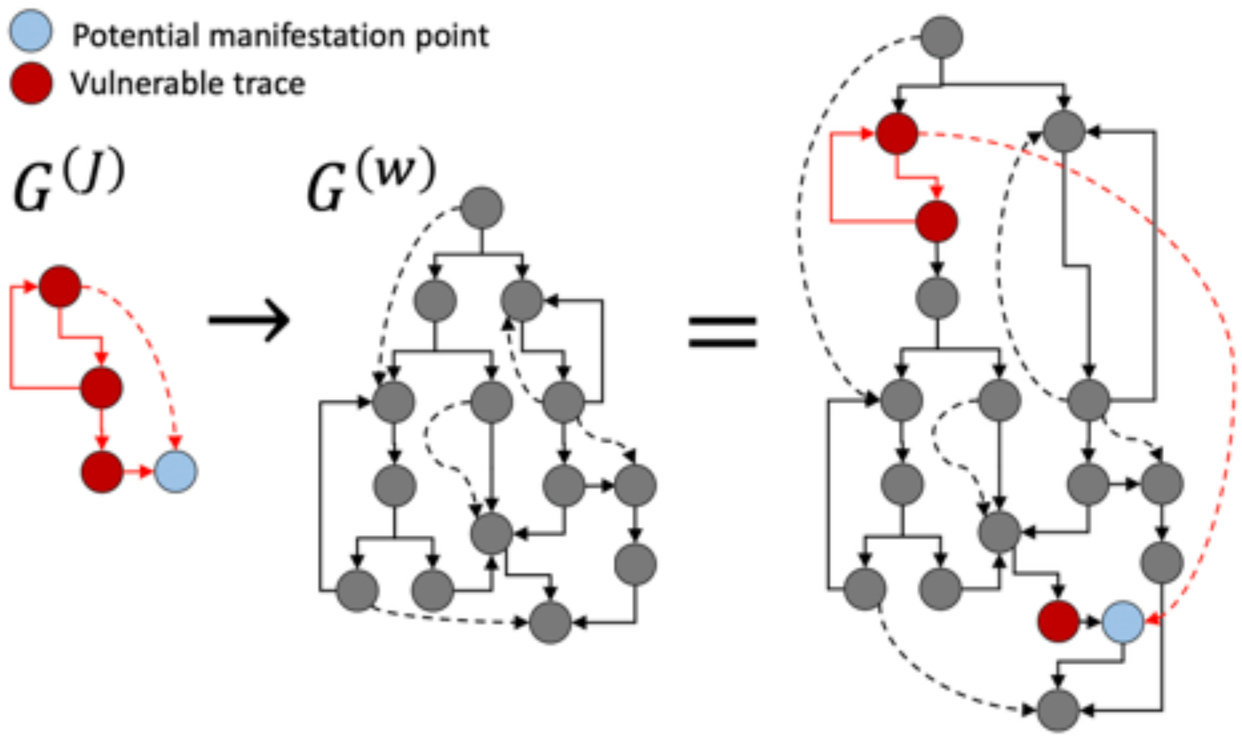
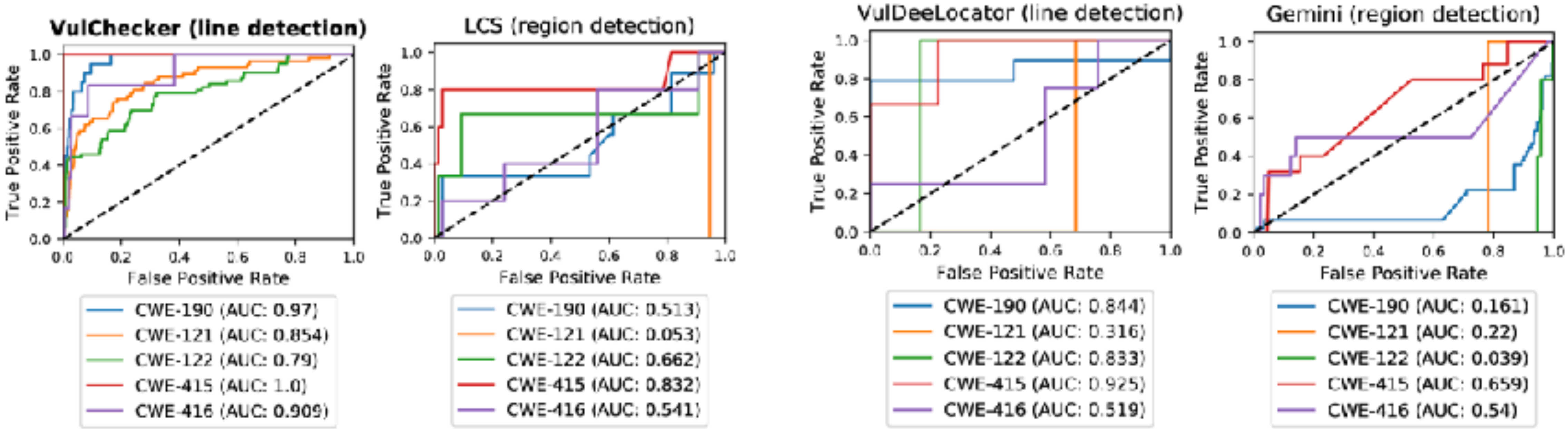


Figure 3: An illustration of an ePDG from the wild $G^{(w)}$ being augmented with a synthetic vulnerability trace from Juliet $G_i^{(J)}$.

Evaluation


VulChecker (line detection)
True Positive Rate
False Positive Rate
CWE-190 (AUC: 0.97)
CWE-121 (AUC: 0.854)
CWE-122 (AUC: 0.79)
CWE-415 (AUC: 1.0)
CWE-416 (AUC: 0.909)
LCS (region detection)
True Positive Rate
False Positive Rate
CWE-190 (AUC: 0.513)
CWE-121 (AUC: 0.053)
CWE-122 (AUC: 0.662)
CWE-415 (AUC: 0.832)
CWE-416 (AUC: 0.541)
VulDeeLocator (line detection)
True Positive Rate
False Positive Rate
CWE-190 (AUC: 0.844)
CWE-121 (AUC: 0.316)
CWE-122 (AUC: 0.833)
CWE-415 (AUC: 0.925)
CWE-416 (AUC: 0.519)
Gemini (region detection)
True Positive Rate
False Positive Rate
CWE-190 (AUC: 0.161)
CWE-121 (AUC: 0.22)
CWE-122 (AUC: 0.039)
CWE-415 (AUC: 0.659)
CWE-416 (AUC: 0.54)

Evaluation (cont.)

Table 3: Baseline comparison against a commercial SAST tool in detecting CVEs in the wild.

	VulChecker @ FPR 0.05			VulChecker @ FPR 0.1			Helix QAC		
	Lines		CVEs	Lines		CVEs	Lines		CVEs
CWE	**TP**	**FP**	**TP**	**TP**	**FP**	**TP**	**TP**	**FP**	**TP**
190	9	55	3	12	112	6	1	2	1
121	7	33	7	9	112	9	4	230	1
122	1	6	1	1	6	1	4	241	1
415	3	0	2	3	0	2	0	5	0
416	4	6	4	6	228	6	0	0	1
Total	**24**	**100**	**17**	**31**	**458**	**24**	**9**	**478**	**4**

Evaluation (cont.)

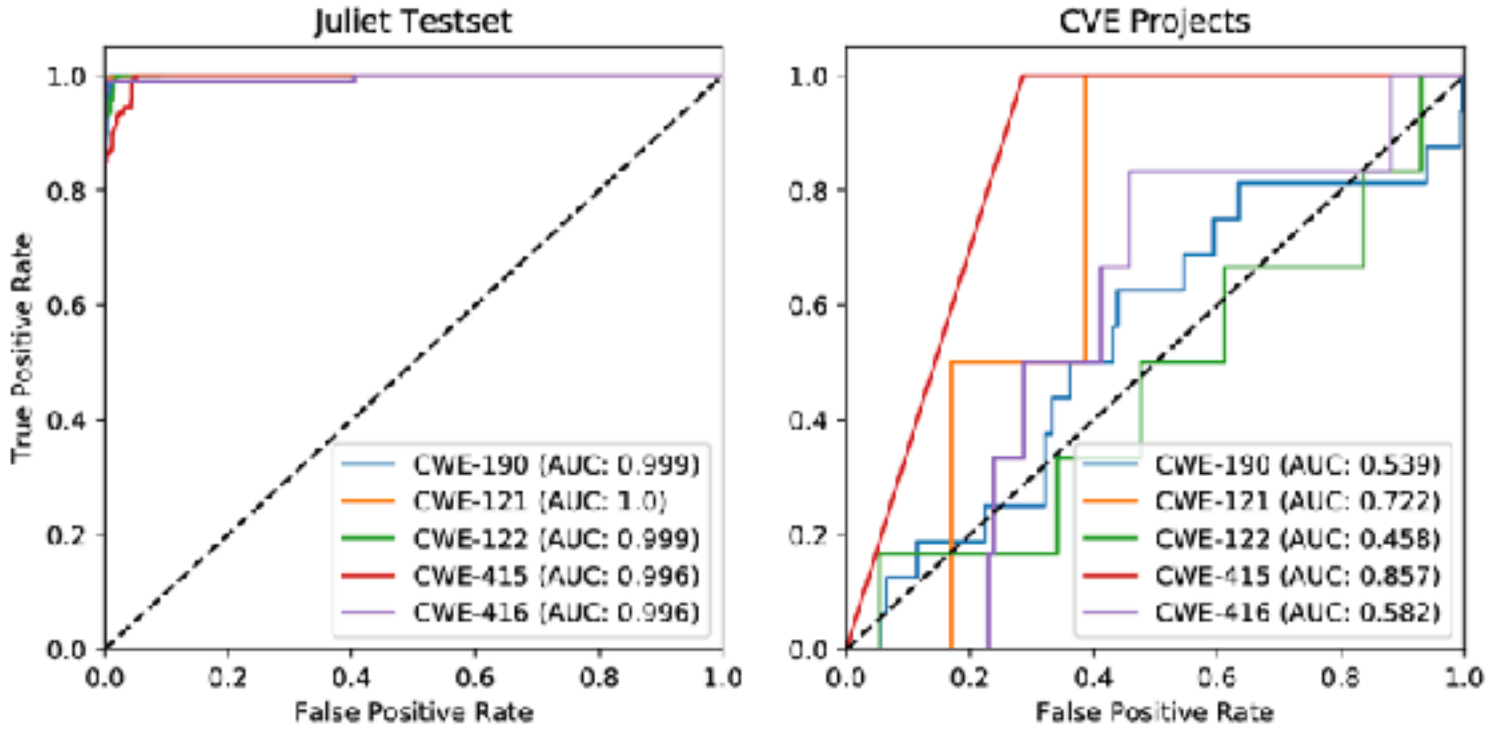


Figure 6: Performance of VulChecker when trained on synthetic data, then either tested on synthetic (left) or tested on real data (right).

Conclusion

- VulChecker precisely locates vulnerabilities in source code down to the exact instruction.
- Classifies vulnerabilities according to the Common Vulnerabilities and Exposures (CVE) taxonomy.
- Employs a novel data augmentation technique to enrich the training dataset and enhance generalization ability.
- Achieves near-zero false positives in vulnerability detection, outperforming commercial tools.
- VulChecker successfully detects a previously unknown zero-day vulnerability, highlighting its ability to identify novel vulnerabilities.

Acknowledgments

- [VulChecker] VulChecker: Graph-based Vulnerability Localization in Source Code, Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, Usenix Security 2023.
- [Alves] Program Slicing. SwE 455, Alves, E., Federal University of Pernambuco, 2015.