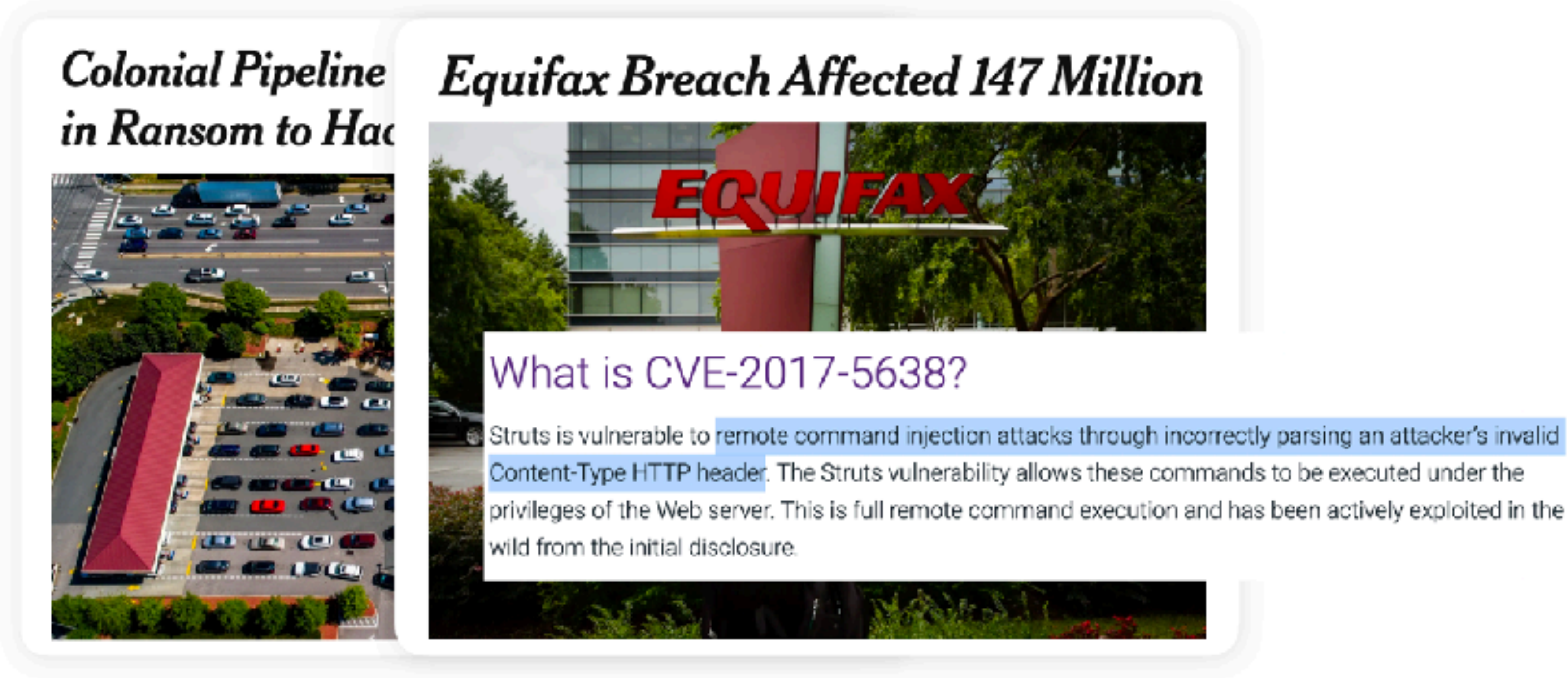
CE 815 – Secure Software Systems

ML-Based Vulnerability Detection Methods (Devign)

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.

Vulnerabilities



Colonial Pipeline
in Ransom to Hac

Equifax Breach Affected 147 Million

What is CVE-2017-5638?

Struts is vulnerable to remote command injection attacks through incorrectly parsing an attacker's invalid Content-Type HTTP header. The Struts vulnerability allows these commands to be executed under the privileges of the Web server. This is full remote command execution and has been actively exploited in the wild from the initial disclosure.

Literature Background

- Manual Code Auditing
- Static Analysis Tools
- Dynamic Analysis Tools

Manual Code Auditing

- Advantages:
  - Thorough and comprehensive analysis of code
  - Ability to detect complex vulnerabilities that may elude automated tools
  - Deep understanding of the codebase and its context
- Limitations:
  - Time-consuming and labor-intensive process
  - Prone to human error and inconsistencies
  - Difficult to scale for large codebases and complex systems

[ChatGPT]

Static Analysis Tools

- Types:
  - Code scanners: Examine source code for potential vulnerabilities and coding errors
  - Lint tools: Identify stylistic issues and potential code defects
  - Data flow analysis: Tracks data flow through code to detect potential vulnerabilities
- Techniques:
  - Lexical analysis: Breaks code into tokens and examines their usage patterns
  - Syntax analysis: Checks code adherence to programming language rules
  - Semantic analysis: Analyzes the meaning and intent of code constructs
- Effectiveness:
  - Effective for detecting a wide range of vulnerabilities
  - Can be integrated into the software development lifecycle (SDLC)
  - Scalable for large codebases and complex systems

[ChatGPT]

Dynamic Analysis Tools

- Principles:
  - Examines software behavior during execution
  - Identifies vulnerabilities by observing how the software responds to various inputs or scenarios
- Approaches:
  - Fuzz testing: Generates random or unexpected inputs to trigger unexpected behavior
  - Symbolic execution: Analyzes code by symbolically representing inputs and variables
  - Runtime monitoring: Tracks program execution and detects anomalous behavior
- Applications:
  - Effective for detecting vulnerabilities that manifest during runtime
  - Complements static analysis by providing insights into dynamic behavior
  - Can be used to test software in various deployment environments

[ChatGPT]

Rise of Machine Learning, Why?

- Improved Accuracy:
  - Machine learning models can discern intricate patterns and subtle anomalies, enhancing the accuracy of vulnerability detection
- Adaptability:
  - ML models can adapt to emerging threats without manual intervention, providing a proactive defense mechanism
- Automation:
  - Automation in the detection process reduces the reliance on predefined signatures, enabling the identification of novel vulnerabilities

[ChatGPT]

Learning Flow for Vulnerability Detection

- Data Collection and Preparation:
  - Preprocessing, feature extraction, and representation
- Model Training:
  - Choosing an appropriate architecture, optimization algorithms, and hyperparameter tuning
- Model Evaluation:
  - Performance metrics, generalization, and robustness

[ChatGPT]

Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks

NeurIPS 2019

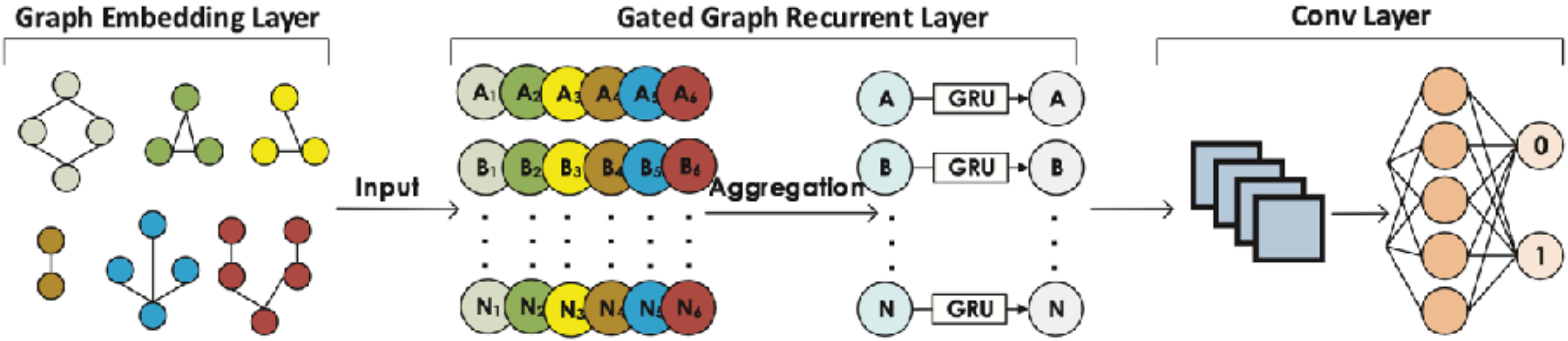
Introduction

- Vulnerability detection:
  - Crucial for cyber security
  - Challenging and requires specialized expertise
  - Traditional approach: Manually-defined vulnerability patterns
  - Limitations: Tedious to create, difficult to maintain

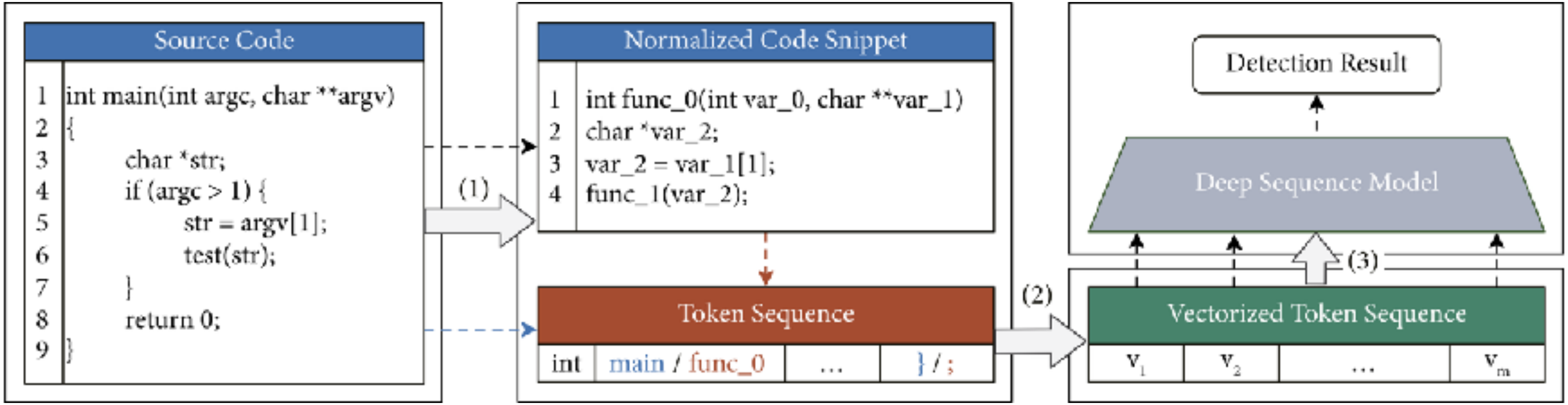
Problem

- Develop an automated approach to learn vulnerability patterns from code
- Leverage graph neural networks (GNNs) to extract comprehensive program semantics
- Design a model that effectively identifies vulnerable functions without manual feature engineering

Workflow


Graph Embedding Layer
Gated Graph Recurrent Layer
Conv Layer
Input
Aggregation
A1 A2 A3 A4 A5 A6
B1 B2 B3 B4 B5 B6
N1 N2 N3 N4 N5 N6
A
B
N
GRU
0
1

Code Sequence Representation



```
int main(int argc, char **argv)
{
        char *str;
        if (argc > 1) {
                str = argv[1];
                test(str);
        }
        return 0;
}
```

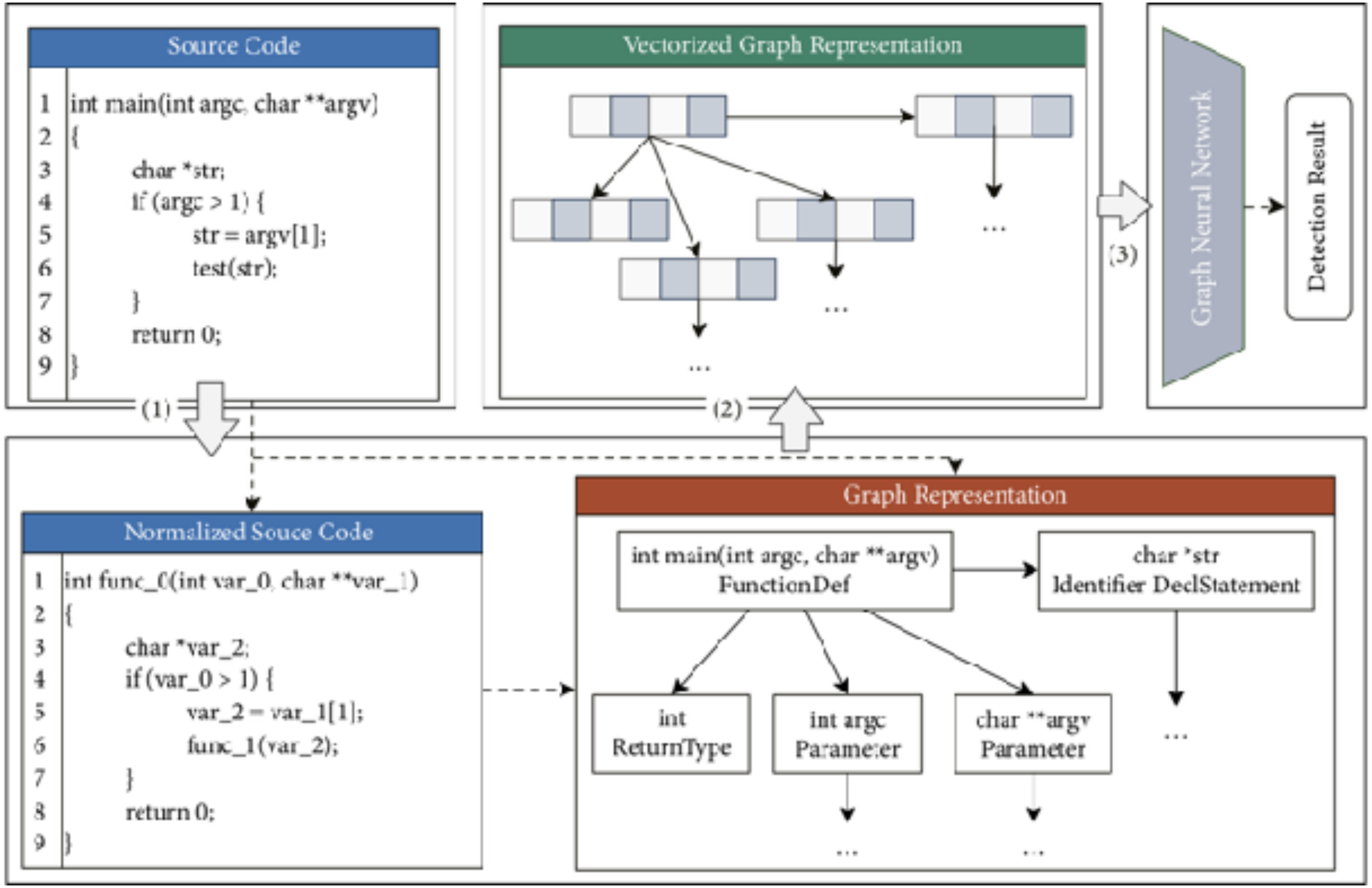
```
int func_0(int var_0, char **var_1)
char *var_2;
var_2 = var_1[1];
func_1(var_2);
```

[Bolun]

Code Graph Representation

- Enhanced Accuracy
  - Code graphs capture relationships between program elements, providing a more accurate representation of code for vulnerability detection.
- Improved Scalability
  - Code graphs are efficiently represented and processed using graph algorithms, enabling effective analysis of large codebases.
- Enhanced Flexibility
  - Code graphs can be easily modified to represent different aspects of code, adapting to various cybersecurity tasks.

Code Graph Representation (cont.)



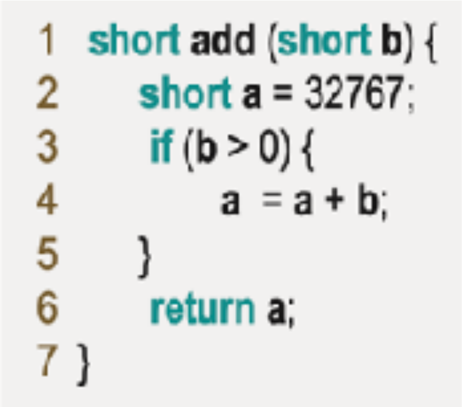
```
int main(int argc, char **argv)
{
        char *str;
        if (argc > 1) {
                str = argv[1];
                test(str);
        }
        return 0;
}
```

```
int func_0(int var_0, char **var_1)
{
        char *var_2;
        if (var_0 > 1) {
                var_2 = var_1[1];
                func_1(var_2);
        }
        return 0;
}
```

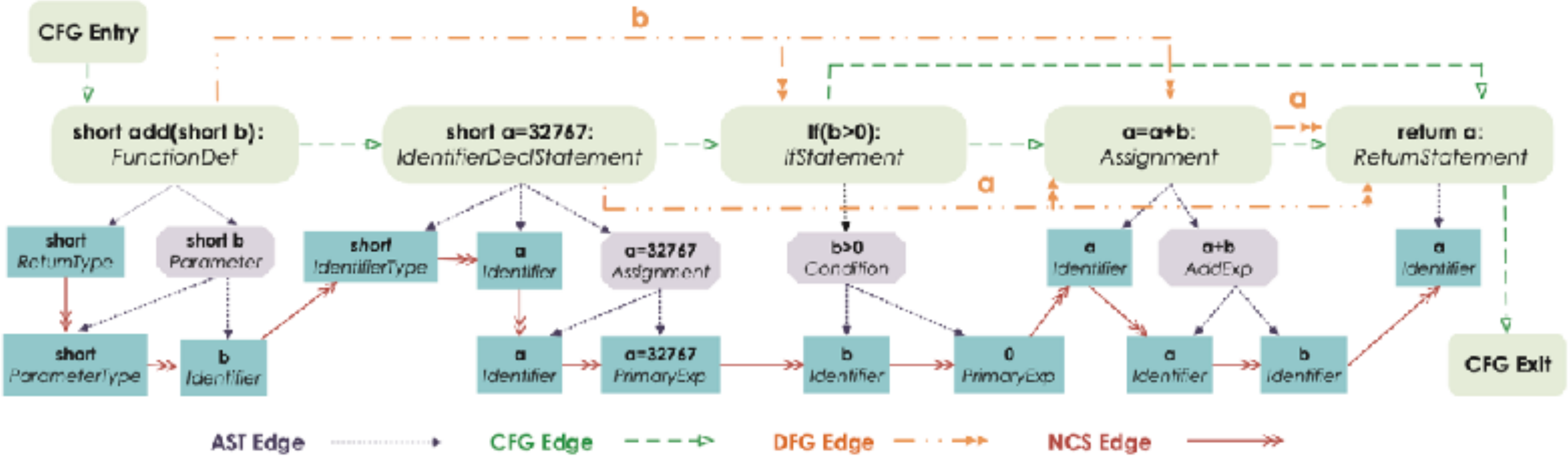
Code Graph Representation (cont.)

- Abstract Syntax Tree (AST)
- Control Flow Graph (CFG)
- Data Flow Graph (DFG)
- Natural Code Sequence (NCS)
  - In order to encode the natural sequential order of the source code, NCS edges are used to connect neighboring code tokens in the ASTs. The main benefit with such encoding is to reserve the programming logic reflected by the sequence of source code

Code Graph Representation (cont.)

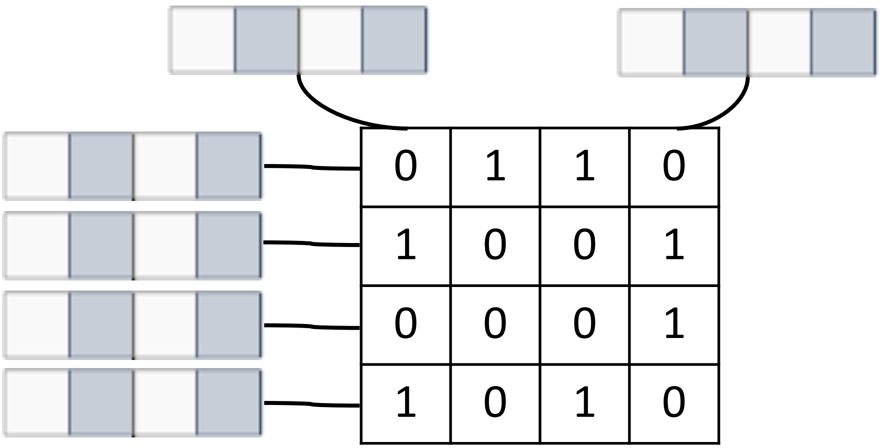


```
short add (short b) {
    short a = 32767;
     if (b > 0) {
          a  = a + b;
     }
     return a;
}
```

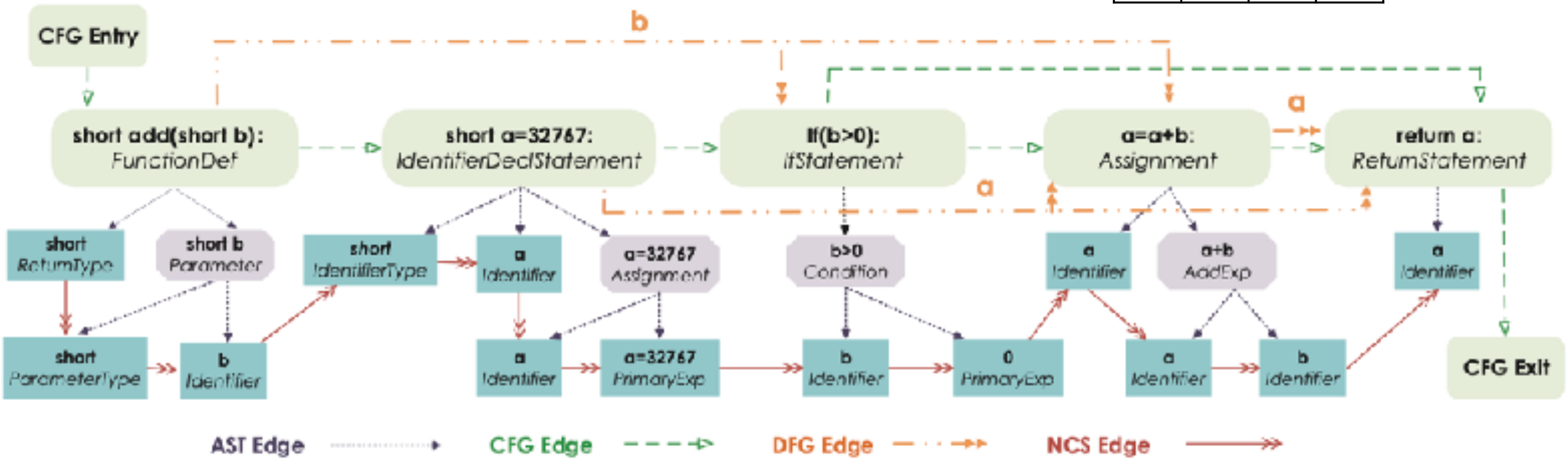


Graph Embedding

- Word2Vec
  - Content | Type

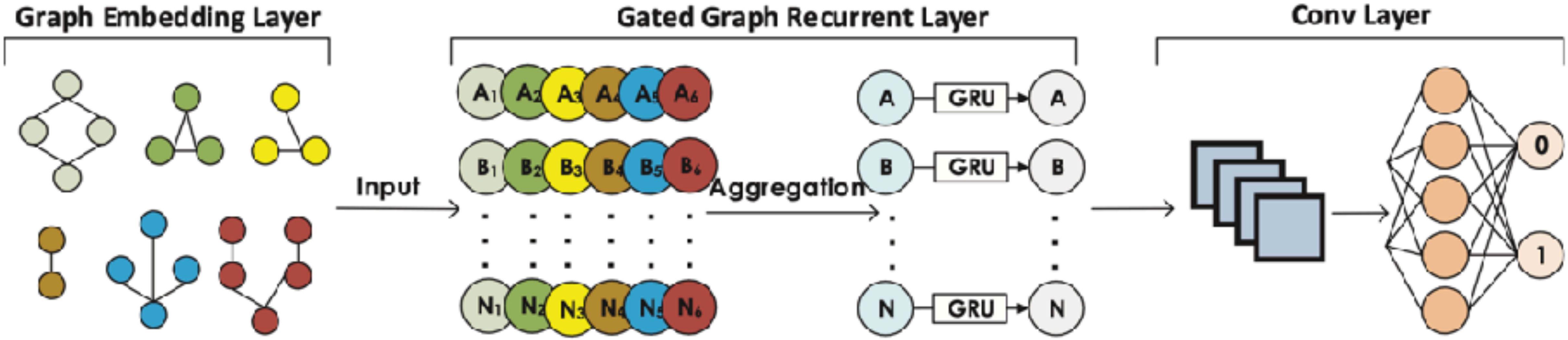


0	1	1	0
1	0	0	1
0	0	0	1
1	0	1	0

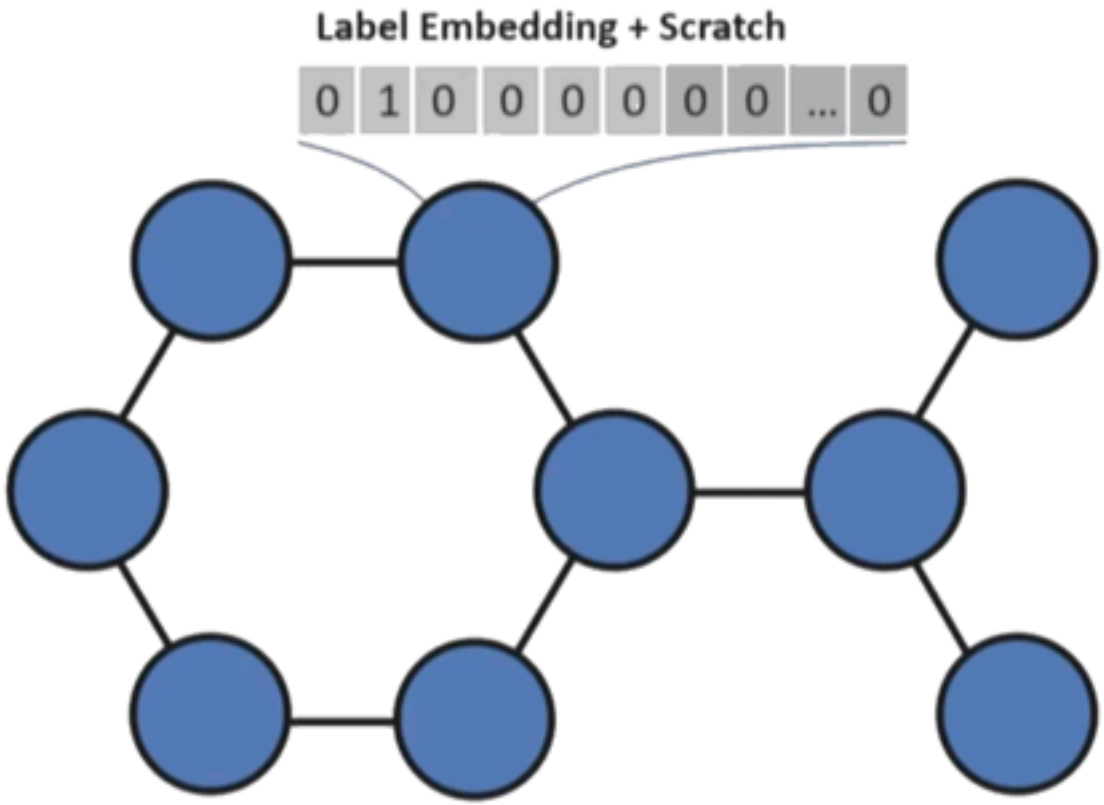


Model

- Learn embeddings for each node, edge, graph
- Message passing
- Using Gated GNN



GNN

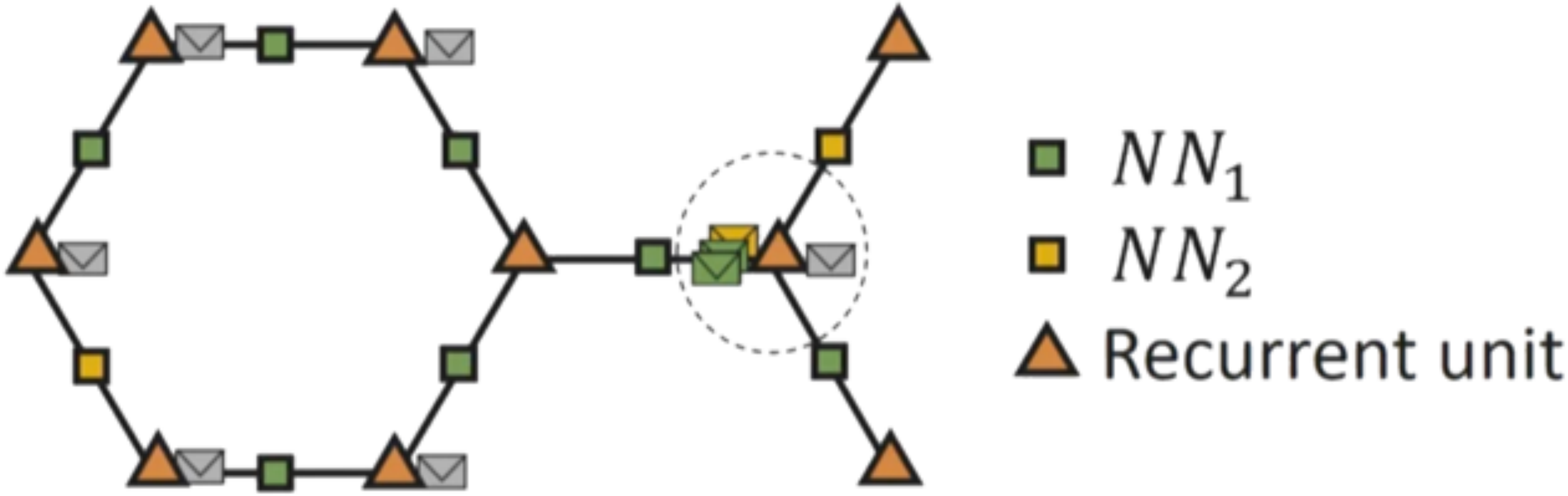


Label Embedding + Scratch

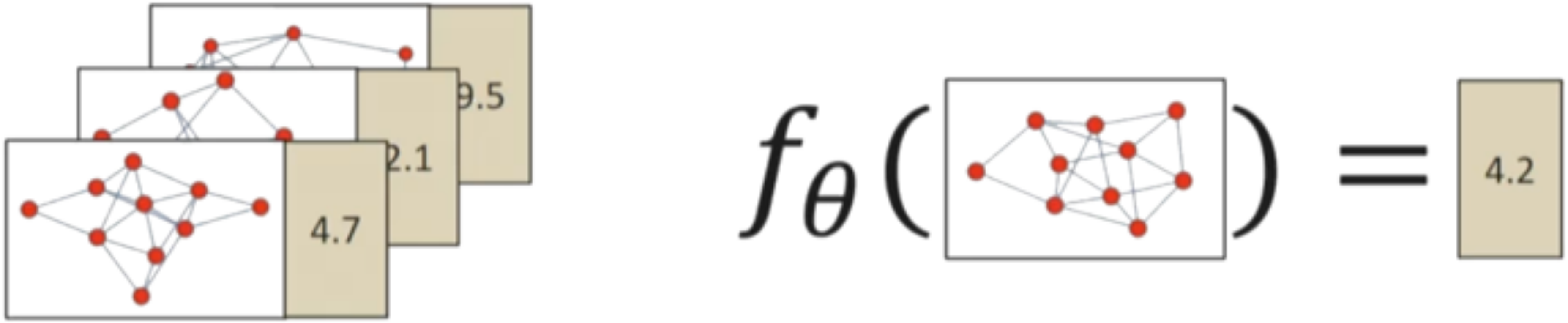
0 1 0 0 0 0 0 0 ... 0

[Gaunt]

GNN (cont.)



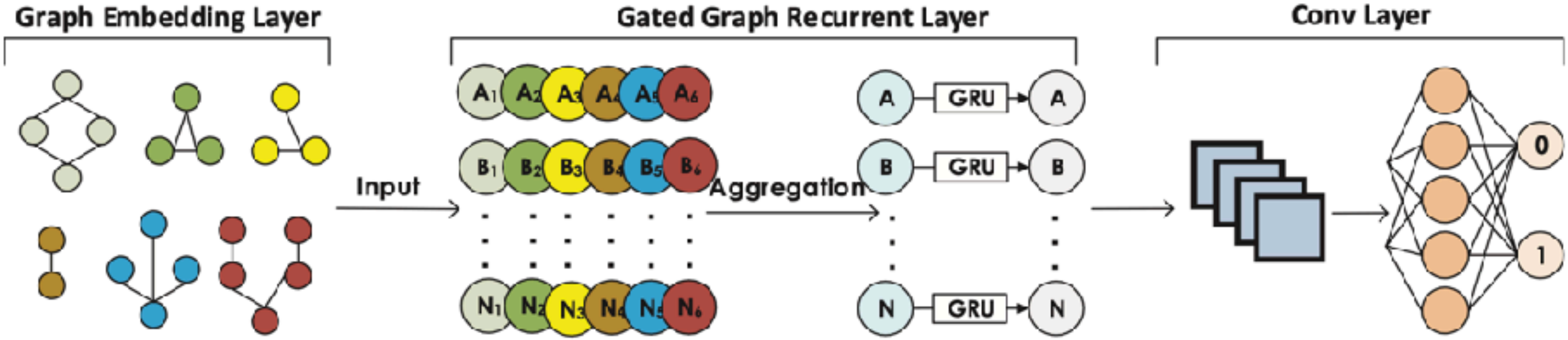
GNN (cont.)



[Gaunt]

Model

- Learn embeddings for each node, edge, graph
- Message passing
- Using Gated GNN



Evaluation

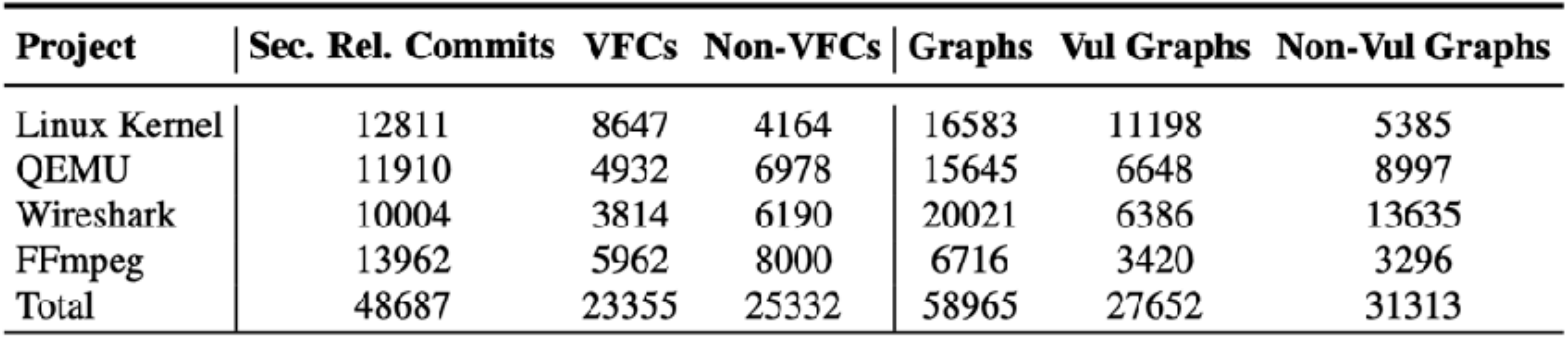
- **Q1:** How does our Devign compare to the other learning based vulnerability identification methods?
- **Q2**: Can Devign learn from each type of the code representations (e.g., a single-edged graph with one type of information)? And how do the Devign models with the composite graphs (e.g., all types of code representations) compare to each of the single-edged graphs?
- **Q3:** Can Devign have a better performance compared to some static analyzers in the real scenario where the dataset is imbalanced with an extremely low percentage of vulnerable functions?
- **Q4:** How does Devign perform on the latest vulnerabilities reported publicly through CVEs?

Dataset

- Data Collection:
  - Four diversified large-scale open-source C projects
  - Manually labeled commits as security/non-security, from which vulnerable and non-vulnerable functions are labeled
- Data Preprocessing:
  - Code parsing to extract AST, CFG, and PDG graphs using Joern
    - Two many different type of edges, limit to 3 types
    - LastRead (DFG_R), LastWrite (DFG_W), and ComputedFrom (DFG_C)

Dataset

Table 1: Data Sets Overview



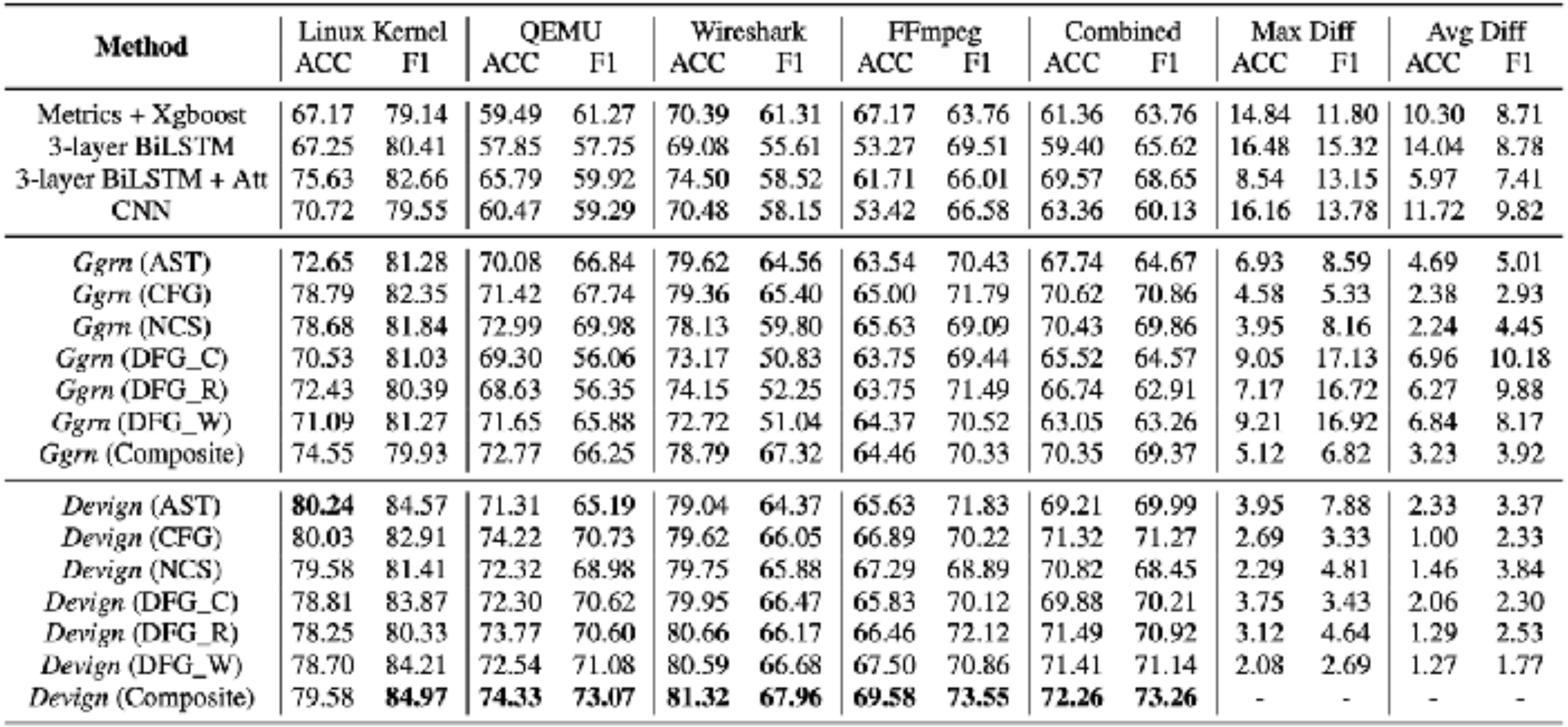
Project	Sec. Rel. Commits	VFCs	Non-VFCs	Graphs	Vul Graphs	Non-Vul Graphs
Linux Kernel	12811	8647	4164	16583	11198	5385
QEMU	11910	4932	6978	15645	6648	8997
Wireshark	10004	3814	6190	20021	6386	13635
FFmpeg	13962	5962	8000	6716	3420	3296
Total	48687	23355	25332	58965	27652	31313

Evaluation

- **Metrics + Xgboost**
  - Collect totally 4 complexity metrics and 11 vulnerability metrics for each function using Joern, and utilize Xgboost for classification
- **3-layer BiLSTM**
  - Treats the source code as natural languages and input the tokenized code into bidirectional LSTMs with initial embeddings trained via Word2vec
- **3-layer BiLSTM + Att**
  - With attention mechanizm
- **CNN**
  - Takes source code as natural languages and utilizes the bag of words to get the initial embeddings of code tokens, and then feeds them to CNNs to learn.

Evaluation (cont.)

Table 2: Classification accuracies and F1 scores in percentages: The two far-right columns give the maximum and average relative difference in accuracy/F1 compared to *Devign* model with the composite code representations, i.e., *Devign* (Composite).

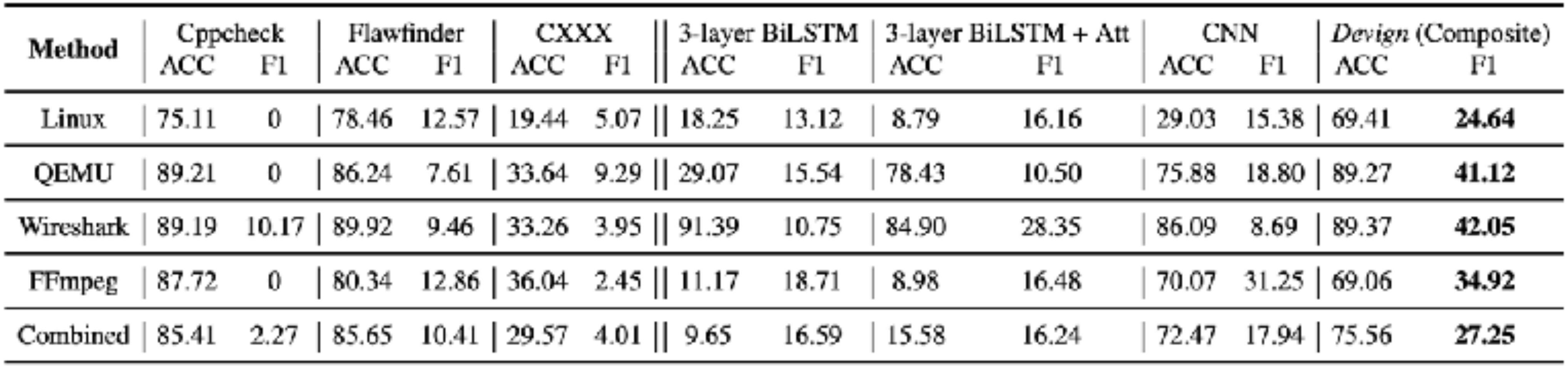


Method	Linux Kernel		QEMU		Wireshark		FFmpeg		Combined		Max Diff		Avg Diff	
	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1
Metrics + Xgboost	67.17	79.14	59.49	61.27	70.39	61.31	67.17	63.76	61.36	63.76	14.84	11.80	10.30	8.71
3-layer BiLSTM	67.25	80.41	57.85	57.75	69.08	55.61	53.27	69.51	59.40	65.62	16.48	15.32	14.04	8.78
3-layer BiLSTM + Att	75.63	82.66	65.79	59.92	74.50	58.52	61.71	66.01	69.57	68.65	8.54	13.15	5.97	7.41
CNN	70.72	79.55	60.47	59.29	70.48	58.15	53.42	66.58	63.36	60.13	16.16	13.78	11.72	9.82
Ggrn (AST)	72.65	81.28	70.08	66.84	79.62	64.56	63.54	70.43	67.74	64.67	6.93	8.59	4.69	5.01
Ggrn (CFG)	78.79	82.35	71.42	67.74	79.36	65.40	65.00	71.79	70.62	70.86	4.58	5.33	2.38	2.93
Ggrn (NCS)	78.68	81.84	72.99	69.98	78.13	59.80	65.63	69.09	70.43	69.86	3.95	8.16	2.24	4.45
Ggrn (DFG_C)	70.53	81.03	69.30	56.06	73.17	50.83	63.75	69.44	65.52	64.57	9.05	17.13	6.96	10.18
Ggrn (DFG_R)	72.43	80.39	68.63	56.35	74.15	52.25	63.75	71.49	66.74	62.91	7.17	16.72	6.27	9.88
Ggrn (DFG_W)	71.09	81.27	71.65	65.88	72.72	51.04	64.37	70.52	63.05	63.26	9.21	16.92	6.84	8.17
Ggrn (Composite)	74.55	79.93	72.77	66.25	78.79	67.32	64.46	70.33	70.35	69.37	5.12	6.82	3.23	3.92
Devign (AST)	**80.24**	84.57	71.31	65.19	79.04	64.37	65.63	71.83	69.21	69.99	3.95	7.88	2.33	3.37
Devign (CFG)	80.03	82.91	74.22	70.73	79.62	66.05	66.89	70.22	71.32	71.27	2.69	3.33	1.00	2.33
Devign (NCS)	79.58	81.41	72.32	68.98	79.75	65.88	67.29	68.89	70.82	68.45	2.29	4.81	1.46	3.84
Devign (DFG_C)	78.81	83.87	72.30	70.62	79.95	66.47	65.83	70.12	69.88	70.21	3.75	3.43	2.06	2.30
Devign (DFG_R)	78.25	80.33	73.77	70.60	80.66	66.17	66.46	72.12	71.49	70.92	3.12	4.64	1.29	2.53
Devign (DFG_W)	78.70	84.21	72.54	71.08	80.59	66.68	67.50	70.86	71.41	71.14	2.08	2.69	1.27	1.77
Devign (Composite)	79.58	**84.97**	**74.33**	**73.07**	**81.32**	**67.96**	**69.58**	**73.55**	**72.26**	**73.26**	-	-	-	-

Evaluation

- **10% vulnerable data**

Table 3: Classification accuracies and F1 scores in percentages under the real imbalanced setting



Method	Cppcheck		Flawfinder		CXXX		3-layer BiLSTM		3-layer BiLSTM + Att		CNN		*Devign* (Composite)	
	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1
Linux	75.11	0	78.46	12.57	19.44	5.07	18.25	13.12	8.79	16.16	29.03	15.38	69.41	**24.64**
QEMU	89.21	0	86.24	7.61	33.64	9.29	29.07	15.54	78.43	10.50	75.88	18.80	89.27	**41.12**
Wireshark	89.19	10.17	89.92	9.46	33.26	3.95	91.39	10.75	84.90	28.35	86.09	8.69	89.37	**42.05**
FFmpeg	87.72	0	80.34	12.86	36.04	2.45	11.17	18.71	8.98	16.48	70.07	31.25	69.06	**34.92**
Combined	85.41	2.27	85.65	10.41	29.57	4.01	9.65	16.59	15.58	16.24	72.47	17.94	75.56	**27.25**

Conclusion

- Devign Key Contributions:
  - Proposed a novel Conv module to efficiently extract useful features for graph-level classification
  - Employed a rich set of code semantic representations for comprehensive program semantics learning
  - Demonstrated significant performance improvement over state-of-the-art methods
  - Achieved an average of 10.51% higher accuracy and 8.68% F1 score
  - Successfully applied to identify vulnerabilities in real-world software projects

Acknowledgments

- [Devign] Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks, Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, NeurIPS 2019.
- [CMSC818I] CMSC818I: Advanced Topics in Computer Systems; Large Language Models, Security, and Privacy, Chen, Y., UMD, 2023.
- [ChatGPT] Content created with the help of OpenAI’s ChatGPT.
- [Bolun] Code vulnerability detection based on deep sequence and graph models: A survey, Wu, Bolun, and Futai Zou, Security and Communication Networks, 2022.
- [Gaunt] Graph neural networks: Variations and applications, Gaunt, A., Youtube, 2016. Retrieved from https://m.youtube.com/watch?v=cWIeTMklzNg